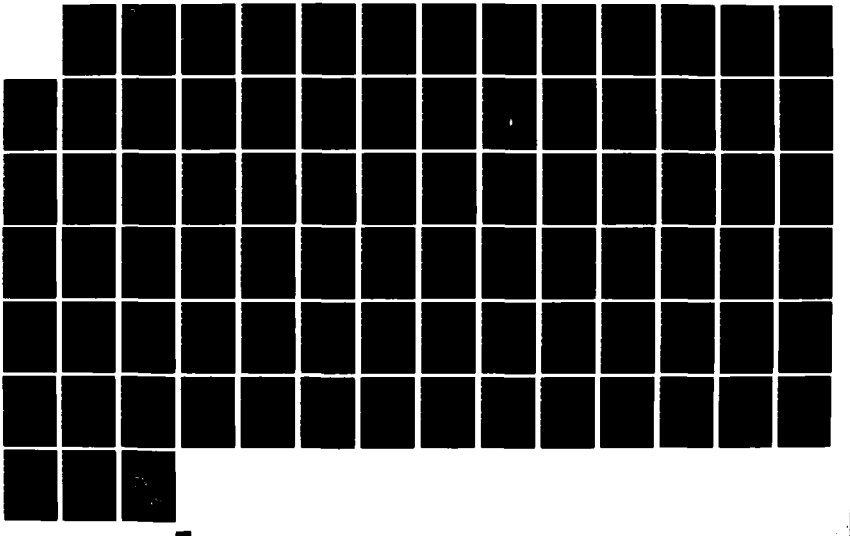
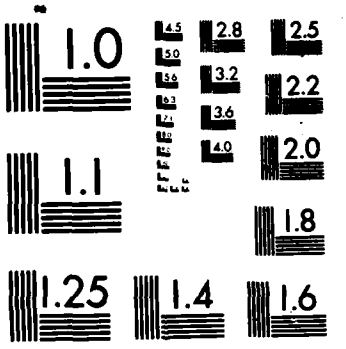
AD-A165 540 INTRODUCTION TO DIGITAL LOGIC SYSTEMS FOR ENERGY MONITORING AND CONTROL SYSTEMS(U) ARMY ENGINEER DIV HUNTSVILLE AL MAY 85 HNDSP-85-ED-ME 1/1

UNCLASSIFIED F/G 10/3 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Army Corps
:ngineers
sville Division

INTRODUCTION TO DIGITAL LOGIC SYSTEMS FOR ENERGY MONITORING AND CONTROL SYSTEMS

HNDSP 85-107-ED-ME
MAY 1985

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER HNDSP-85-107-ED-ME	2. GOVT ACCESSION NO. AD-A165540	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* INTRODUCTION TO DIGITAL LOGIC SYSTEMS FOR ENERGY MONITORING AND CONTROL SYSTEMS		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s)		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS U.S. Army Engineer Division, Huntsville P.O. Box 1600 Huntsville, Alabama 35807-4301		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE MAY, 1985 13. NUMBER OF PAGES 82 PAGES
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE: DISTRIBUTION UNLIMITED.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

DIGITAL SYSTEMS ENERGY MANAGEMENT MONITORING ARMY CORPS OF ENGINEERS

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

RECENT ADVANCES IN THE STATE-OF-THE-ART OF DIGITAL ELECTRONIC TECHNOLOGY HAVE MADE FEASIBLE THE CONCEPT OF DISTRIBUTED DATA PROCESSING (DDP) IN ENERGY MONITORING AND CONTROL SYSTEMS (EMCS). THESE ADVANCES WERE BROUGHT ABOUT LARGELY DUE TO THE ADVENT OF MICRO-MINIATURIZATION OF ELECTRONIC INTEGRATED CIRCUIT COMPONENTS AND THE DEVELOPMENT OF THE MICROPROCESSOR, A DISCRETE SEMICONDUCTOR DEVICE WITH MANY OF THE CAPABILITIES OF THE CENTRAL PROCESSING UNIT (CPU) OF THE FAMILIAR MAINFRAMES OR MINICOMPUTERS, BUT VASTLY REDUCED IN

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

SIZE AND COST. IT IS NOW POSSIBLE TO DESIGN A RELIABLE EMCS CONSISTING OF A CENTRAL COMPUTER CONTROL SYSTEM AND REMOTE MICROCOMPUTERS WHICH CAN PERFORM MANY FUNCTIONS INDEPENDENT OF THE CENTRAL SYSTEM. WHEN PROPERLY APPLIED, THESE EMCS INSTALLATIONS CAN AID SIGNIFICANTLY IN CONSERVING OUR DWINDLING SOURCES OF ENERGY.

Keywords: Logic design; computer logic; computer architecture; assembly languages; high level languages; interfacing; microprocessors.

EHCS

INTRODUCTION TO DIGITAL LOGIC SYSTEMS

MAY 1985

CONTENTS

CONTENTS

Section I. PURPOSE

I-1. Definition. Digital logic systems encompass a broad class of electronic hardware which includes the digital computer. Computers utilize digital logic to perform various simple arithmetic or logical operations in step-by-step sequences at extremely high speeds in order to solve complex problems. They also may perform logical and arithmetic operations for controlling or monitoring real time processes. Digital logic operations are generally carried out by electronic devices which may exist in either of two possible states expressed as "on" and "off". Some digital logic devices are dedicated to a specific purpose, whereas computers are programmable; that is, they may be adapted to a wide variety of tasks. A general purpose digital computer is virtually unlimited in its ability to solve problems and control processes. However, it is virtually useless without the addition of system software consisting of a series of logical operations written in mathematical form to direct the basic operations of the machine. This software, sometimes referred to as the system monitor, reduces the task of programming from near impossibility to relative ease. A digital computer, in conjunction with other digital hardware components and electromechanical devices, forms a system which can automate numerous types of processes.

I-2. Application to EMCS.

a. Large buildings or building complexes are heavy consumers of energy. They are therefore prime candidates for energy conservation. This reduction in energy consumption could be performed by manually operating the building utility equipment systems at their optimum efficiency points and by carefully observing schedules to enable the shutdown of any equipment not required, such as lights, fans, and chillers when personnel are not present. However, it becomes nearly impossible for a human operator to accomplish this task when buildings are large and complex. In addition, it is more difficult for a human operator to analyze data trends and use this information to optimize the starting and stopping of equipment. Digital computers are ideal for automated monitoring and control of building energy systems.

b. Recent advances in the state-of-the-art of digital electronic technology have made feasible the concept of distributed data processing (DDP) in Energy Monitoring and Control Systems (EMCS). These advances were brought about largely due to the advent of micro-miniaturization of electronic integrated circuit components and the development of the microprocessor, a discrete semiconductor device with many of the capabilities of the central processing unit (CPU) of the familiar mainframes or minicomputers, but vastly reduced in size and cost. It is now possible to design a reliable EMCS consisting of a central computer control system and remote microcomputers which can perform many functions independent of the central system. When properly applied, these EMCS installations can aid significantly in conserving our dwindling sources of energy.

c. The following sections present the basic concepts of digital logical systems, computer architecture, and the technological advances which have led to current EMCS philosophy.

Section II. REVIEW OF NUMBER SYSTEMS

II-1. Base 10 number system.

a. Due to physiological considerations, mankind has come to base the way he counts on the powers of ten; thus, we are said to have a base 10 (decimal) number system. In actuality, there is no particular reason or mathematical advantage in using 10 as a base for arithmetic. Any particular number would do, and humans could learn to perform arithmetic as well with any of them if taught to do so from the beginning. In point of fact, numbers are merely symbolic representations of the concept of "quantity", which may take on an infinite number of values.

b. Mankind has long attempted to apply physical laws of nature in the form of mechanical or electrical devices to aid in making arithmetic calculations. The prime criteria for these devices are improvements in speed and accuracy. Ultimately, mechanical calculators were built that could perform arithmetic a good deal faster than humans. These machines were designed to use base 10 numbers which made them ideal for the way humans do arithmetic. However, as useful as these mechanical calculators have proven to be, they have two major defects:

(1) They are relatively slow.

(2) They cannot be "programmed" to solve complex and unique problems.

c. It was observed that, through the use of electricity, calculators could be made to operate at vastly higher speeds, limited only by the speed of light. In addition, it was shown that programming these electronic calculators would be possible. It was observed that working with base 10 numbers on electronic calculators is impractical because of the difficulty in defining a correspondence between numbers and electrical properties. The most easily detectable states of a physical property like electricity are "on" and "off". Therefore, calculators are designed to utilize base 2 or "binary" arithmetic. The translation of binary numbers to base 10 can be performed by the computers themselves, in order to be more easily intelligible to humans. To distinguish between number systems, the system base is written as a subscript; for example, 25_{10} is the number 25 in base 10.

II-2. Binary number system. In the binary number system, as well as all other number systems, the concept of "position" is of prime importance. Position refers to the weighting factor assigned to the successive placement of digits to the left of the first number, which is assumed to be in the units position. The familiar base 10 system treats each position as a power of 10. The number 2561 is constructed thusly:

Position	10^3:	10^2:	10^1:	10^0	
Number	2	5	6	1	$= 2\times10^3 + 5\times10^2 + 6\times10^1 + 1\times10^0$

Likewise, the binary number system treats each digit position (or "bit") as an increasing power of 2. The number 13_{10} is constructed in base 2 as follows:

Position 2^3: 2^2: 2^1: 2^0
Number 1 1 0 1 $= 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 = 13_{10}$

a. Binary arithmetic functions. Binary numbers may be added, substracted, multiplied, and divided in the same manner as base 10 numbers, except that there are no extensive tables to remember. In fact, the whole procedure of binary arithmetic is rather simple.

b. Binary addition. For addition, the only facts to consider are that 0 plus 0 equal 0, 1 plus 0 equals 1, and 1 plus 1 equals 0 with 1 to be carried to the left. In mathematical form:

$0 + 0 = 0$
$1 + 0 = 0 + 1 = 1$
$1 + 1 = 10$ (where the left digit is a carry)

With the above simple rules, any number expressed in binary form may be added. As an example, the number 23_{10} and 17_{10} are added below in binary form:

$23 = 2^4 + 2^2 + 2^1 + 2^0 =$ (1 111) : carry / 10111

$+17 = 2^4 + 2^0$ + 10001

40 101000

$101000 = 1 \times 2^5 + 1 \times 2^3 = 32 + 8 = 40_{10}$

c. Binary subtraction. Subtraction is the inverse process of addition; in binary arithmetic, it can be performed in an method analogous to that of base 10 arithmetic. The rules defined for binary subtraction are similar to that of base 10:

$1 - 1 = 0$
$0 - 0 = 0$
$1 - 0 = 1$
$0 - 1 =$ must borrow; or -1
$10 - 1 = 1$ borrow from left digit (10 equals 2_{10})

As an example 27_{10} is subtracted from 39_{10}

$$39 = 2^5 + 2^2 + 2^1 + 2^0 = 100111$$

$$-\frac{27}{12} = 2^4 + 2^3 + 2^1 + 2^0 = \frac{11011}{01100}$$

$$1100 = 2^3 + 2^2 = 12_{10}$$

Digital logic systems, however do not generally perform subtraction as defined above for two reasons:

(1) Difficulties can arise in subtraction as well as addition when dealing with negative numbers unless special care is taken in defining them.

(2) Digital electronic hardware does not always, for various reasons, define a separate "subtract" function. Instead, when a subtraction operation is desired, a number is converted to its complement (negative form) and added to the other number.

d. Ten's complement. In the decimal number system the negative representation of a number, known as the ten's complement, is determined by finding a number, which when added to the absolute value of the desired negative number equals zero (where carries beyond the number of digits of interest are ignored.) To illustrate this, the ten's complement representation of -3 is considered, where the maximum number capable of being represented by the calculating device is 6 digits (999999):

$$\begin{array}{rl} |-3| = & 3 \\ & 3 \\ + & 999997 \\ \hline 1 & 000000 \end{array}$$

carried to 7th place & ignored

The ten's complement of -3 is thus 999997 for 6 digit numbers. It can be shown that a simple method of finding the ten's complement of any negative number is to subtract the absolute value of the number from N digits of 9's and add 1 to the result. The usefulness of ten's complement representation is that the subtraction process can be reduced to an addition process without worrying about signs. Any ten's complement number with a 5 or higher in the most significant digit (left most) place is assumed to be a negative number.

A minor disadvantage of this representation is that it limits the number of positive numbers that can be represented for a particular number of digits. Considering ten's complement two digit numbers for instance, positive numbers range between 1 to 49, while negative numbers are in the range from 50 to 99. Since integers are infinite in extent, these limitations can be eliminated merely by considering greater numbers of digits. The preceding discussion can be extended to include decimal fractions as well. Ten's complement arithmetic is not really important for working with base 10 numbers, since humans are used to dealing in signs. However, the application of this concept to binary numbers makes the arithmetic of computers more efficient. Sometimes a nine's complement numerical representation is used for subtraction in computers. It is formed in a manner similar to ten's complement except that a one is not added after subtraction from the nines.

e. Two's complement arithmetic. Application of the above concepts to the binary number system results in what is known as two's complement arithmetic. The application of two's complement is simpler to implement in binary numbers than in base 10 numbers. To form the two's complement of a negative number, the 0's and 1's in the binary number are complemented by interchanging all 0's and 1's and adding 1 to the result. In the example below, the two's complement of 110101 (53_{10}) is formed:

```
Complement  001010
Add 1            1
           ------
           001011   two's complement of 53(10)
```

The number 1011 (11_{10}) can be shown to be the two's complement of 110101, since when it is added to it, the sum is zero (with the 7th digit carried).

```
          110101
            1011
        1 000000
carry
7th digit
```

As in the base 10 number system, half of the total possible numbers in a two's complement representation will be positive; the other half will be negative. If 8 digit binary numbers are used, two's complement representation yields 127 positive numbers and 128 negative numbers, since all numbers in the most significant digit are negative.

f. Multiplication and division. The ability to perform multiplication and division is another essential function to be performed by digital computers. In order to keep computer systems as simple as possible, many machines have not been designed with specific multiply or divide functions. Instead, the ability to perform multiplication and division is accomplished by software (programmed operations) through a series of mathematical and logical operations such as addition or subtraction. More expensive computer systems have included hardware to perform multiply and divide ope ations. This usually results in faster operation of the computer and added simplicity in programming.

g. Binary multiplication. Binary multiplication is performed similarly to decimal multiplication in that the partial product is moved to the left as each successively more significant multiplier digit is used. The multiplication products of binary arithmetic are much simpler than in decimal arithmetic as shown:

$0 \times 0 = 0$
$0 \times 1 = 1 \times 0 = 0$
$1 \times 1 = 1$

As an example, the binary numbers 10001 (17_{10}) and 1011 (11_{10}) are multiplied together:

```
  10001    (17)
x 1011     (11)
 ------
  10001
 10001
00000
+10001
 ------
10111011 = 187(10)
```

Binary multiplication is seen in the above example to be simple; however, it is also very tedious and time-consuming when larger valued numbers are considered. Consequently, binary multiplication is well suited to a computing machine which operates at high speed and is oblivious to tedium.

h. Binary division. Binary division is analogous to decimal division. In the example below, the number 15_{10} is divided by 5_{10} and 11_{10} is divided by 2_{10} using binary representation.

(a)

```
      11
   ------
101)1111
    101
   ----
     101
     101
   ----
       0
```

(b)

```
     101.1
   --------
10)1011.0
   10
   ----
    11
    10
   ----
     10
     10
   ----
      0
```

Example (b) shows binary fractional representation. In the decimal number system, each digit to the right of the decimal point is successively divided by 10 in value. In the binary system, each value to the right of the point is successively divided by 2.

Thus, 101.1 in binary notation is equivalent to 5.5 in decimal numbers.

II-3. Octal number system. An octal number system has a base of 8; therefore, the numbers 0 through 7 make up the system. By itself, the octal number system is of little interest; however, with respect to computers, it provides an easier means of dealing with binary numbers which are clumsy to use and difficult for the programmer to remember. Programmers who work in computer machine language must deal frequently with binary numbers 8 to 32 or more bits in length. Since 3 bits in binary notation can represent a total of 8 numbers, a long binary number can be broken up into groups of these bits starting from the least significant bit, and the octal value can be substituted As an example, the 16 bit number 1001101011011100 is converted to octal notation:

1 : 001 : 101 : 011 : 011 : 100

1 1 5 3 3 4

The octal number 115334 is somewhat easier to deal with and remember than its binary equivalent; thus octal numbers are frequently used by programmers as a shorthand method. Octal numbers are still less recognizable and efficient than decimal numbers; the octal number 1153348 is equivalent to 39644_{10}.

II-4. Hexadecimal number system. Like the octal number system, Hexadecimal numbers are useful as a shorthand notation of binary numbers for programmers. The hexadecimal number system is based on 16; therefore the numbers 0 through 15 make up the system. Since it is desirable that each number in the hexadecimal system consist of one digit each, the two digit numbers 10 through 15 have been replaced with the alphabetical characters A through F. The hexadecimal number system, like the decimal number system and all others, uses

positional dependence with each position to the left being worth 16 times that of its neighbor on the right. The correspondence of hexadecimal numbers to decimal, octal, and binary numbers is shown in Table II-1. Since 4 binary bits can be used to represent the numbers 0 through 15, a long binary number may be broken up into groups of 4 bits and substituted by the hexadecimal values. Taking the same binary number as in the previous example, 39644_{10}, the conversion to Hexa-decimal notation is as follows:

```
1001 : 1010 : 1101 : 1100
  9      A      D      C
```

It can be seen that the hexadecimal number 9ADC is equivalent to 115334_8 or 39644_{10}. Thus, hexadecimal notation is an even more compact representation than the decimal number system, although it is less easily recognizable to those accustomed to the common base 10 system. The choice of which type of notation to use rests with the programmer. However, many manufacturers have written the instruction manuals of their particular computer or microprocessor product about either the octal or hexadecimal number systems. Thus, a working knowledge of both of these number systems is important.

Table II-1. Numerical code equivalents.

DECIMAL	BINARY	OCTAL	HEXADECIMAL	GRAY
0	0	0	0	0000
1	1	1	1	0001
2	10	2	2	0011
3	11	3	3	0010
4	100	4	4	0110
5	101	5	5	0111
6	110	6	6	0101
7	111	7	7	0100
8	1000	10	8	1100
9	1001	11	9	1101
10	1010	12	A	1111
11	1011	13	B	1110
12	1100	14	C	1010
13	1101	15	D	1011
14	1110	16	E	1001
15	1111	17	F	1000
16	10000	20	10	.
17	10001	21	11	.
:	:	:	:	.

II-5. Alphanumeric codes. The preceeding several paragraphs have dealt with number systems useful in performing mathematical operations in a computer. A computer is also capable of performing a variety of other functions, including logic operations, control of processes, and manipulations of alphanumeric data. For that reason, a number of symbolic codes have been developed in an attempt to standardize the representation of all types of data, including non-mathematical information, to make them available to computer systems.

a. ASCII code. The most commonly used code for alphanumeric symbolic data is the ASCII code, an acronym standing for American Standard Code For Information Interchange. The ASCII code represents the numerical digits from 0 through 9, the alphabet and a number of standard symbols such as $, #, ?, etc. The code uses either 6, 7, or 8 binary bits to represent the alphanumeric characters. Only 64 different characters can be represented by 6 binary bits, so codes using more than 6 bits must be used on computer systems which recognize more than 64 characters. The 7 bit ASCII code, representing a maximum of 128 characters, is the most popular code in present use. Eight bit ASCII is a code where the 8th bit is used for parity checks for detecting errors. To eliminate the tedious binary representation of the ASCII code, octal or hexadecimal notation can be used. Table II-2 presents the 6, 7, and 8 bit ASCII codes in their octal equivalent for alphanumeric characters.

TABLE II-2. ASCII, BCD and EBCDIC alphabet codes.

Character	ASCII 6 Bit Octal	ASCII 7 Bit Octal	ASCII 8 Bit Octal W/O Parity	ASCII 8 Bit Octal Even Parity	BCD Octal	EBCDIC Hexadecimal
A	01	101	301	101	61	C1
B	02	102	302	102	62	C2
C	03	103	303	303	63	C3
D	04	104	304	104	64	C4
E	05	105	305	305	65	C5
F	06	106	306	306	66	C6
G	07	107	307	107	67	C7
H	10	110	310	110	70	C8
I	11	111	311	311	71	C9
J	12	112	312	312	41	D1
K	13	113	313	113	42	D2
L	14	114	314	314	43	D3
M	15	115	315	115	44	D4
N	16	116	316	116	45	D5
O	17	117	317	317	46	D6
P	20	120	320	120	47	D7
Q	21	121	321	321	50	D8
R	22	122	322	322	51	D9
S	23	123	323	123	22	E2
T	24	124	324	324	23	E3
U	25	125	325	125	24	E4
V	26	126	326	126	25	E5
W	27	127	327	127	26	E6

b. Baudot code. The Baudot code was created originally for teletype and paper tape devices. It uses five information bits, to represent 32 characters. The binary 1's or 0's are represented as "marks" or "spaces" in a current loop. Since this is insufficient for the 26 letters of the alphabet, 10 numbers and other symbols, a coded character is used to indicate a carriage shift to send or receive the other characters on the upper case. The Baudot code is obsolete; however, some older equipment which is still functioning makes use of this code. The Baudot Alphanumeric equivalents may be found in standard sources.

c. Binary coded decimal (BCD). The binary coded decimal code is not a signal code but rather consists of a number of similar codes. BCD is capable of representing decimal numbers from 0 to 9 by 4 bit binary number groups which are the same as hexadecimal numbers for the same decimal values. The decimal number "9542" would be expressed in its BCD equivalent as a single 16 bit binary number:

9	5	4	2
1001	0101	0100	0010

Note that 1001010101000010 is not numerically equal to 9542_{10}; the computer must be informed that it is dealing with numbers in BCD representation. The BCD code has been extended to cover a total of 64 alphanumeric characters by assigning a two digit decimal code to each one. Other BCD type codes are created by assigning different values to the positions in each 4 bit number other than the usual 8, 4, 2, 1 values. For example, one code uses 7, 4, 2, 1.

d. Extended binary coded decimal (EBCDIC). An extended BCD code known as EBCDIC was introduced by IBM. It has become widely accepted as a coding format because its 8 bit per character structure enables the representation of 256 characters, a number sufficient to accomplish the most demanding tasks.

e. Excess-3 BCD numbers. A representation known as Excess-3 BCD is sometimes used in computer systems because it simplifies the nine's complement format used in subtraction operations. The Excess-3 representation is performed by adding 3 to the decimal digit and converting to the 4 bit binary representation. The nine's complement Excess-3 BCD is formed by interchanging 0's and 1's.

f. Gray code. The Gray code shown in Table II-1 is suitable for process control and monitoring functions because each transition from one number to the next results in a change of only one bit. This minimizes the ambiguity which might be encountered by the controls of a physical device during the momentary transition between two adjacent numbers. Also, data reading errors

are minimized when only one bit changes during adjacent number transitions, whereas in actual binary numbers, the transition of 7 to 8 involves a change in 4 bits (0111 to 1000), with a corresponding increase in the error rate.

g. Parity and error considerations. It is inevitable that a computer system which handles data at high speeds and stores large quantities of data internally will occasionally lose one or more bits of information during its operations. Among the causes of errors besides unreliable hardware are noisy environments, line crosstalk, power fluctuation and thermal variations. The computer has no way of knowing whether the transmission is correct without a specific means of checking the data. The effects of even one erroneous bit of information can range from relative unimportance to complete catastrophe in terms of reliable output or actual computer operation, as will be apparent in the sections dealing with computer architecture and programming.

(1) Parity bits. The simplest method of dealing with potential errors in data involves the concept of parity. In computer parlance, parity refers to the requirement that the total number of positive bits (values of 1) in the data bit information group be either even or odd. Due to the architectural considerations of a computer, data is usually handled internally in groups of a constant number of bits, known as a "word". Also, the use of alphanumeric codes such as ASCII has given rise to a bit group known as a "byte" which normally is considered to consist of 8 bits (although it may sometimes be considered as 6 bits). External transmission into and out of the computer may consist of one or more bytes in a contiguious block of information. Parity is applied to words or bytes in a variety of ways. For instance, 8 bit ASCII data can be given even or odd parity by replacing a single binary "0" with a "1" to some of the coded characters. This change is made to the most significant bit or in such a way as not to upset the codes, which is possible because only 128 of a possible 256 characters are represented by the 8 bit ASCII code. The decision to use either even or odd parity is made by the systems programmer or manufacturer. Data is sometimes stored in computer memories which have extra parity bits to ensure against internal errors. This also helps indicate whether portions of the memory are experiencing failures. Parity is also applied to mass storage media such as magnetic tapes and disks to decrease the probability of errors. This will be discussed in more detail in Section IV. A weakness in the parity concept is that parity can be used to detect errors in a single bit only. If two or more bits are in error, the correct parity may still be observed with erroneous data. Errors made by the programmer are, of course, not detectable by computer hardware or software, except language syntax errors as discussed in Section V.

(2) Hamming codes. More advanced methods of error detection and correction involve the use of sophisticated mathematical approaches implemented via programs or hardware. One such technique make use of Hamming codes, which are capable of detecting and correcting single bit and double bit errors. The penalty to be paid for detection and correction of errors is that extra bits must be transmitted containing coded information generated in accordance with the data. The required number of error checking bits rises

in correspondence to the number of simultaneous bits corrected. In one such Hamming code method, each 4 bit group of data (referred to as a "Nybble") is attached to four bits of Hamming code data. Since there are only 16 possible data Nybbles, the resulting 16 bytes of data are unique among the 256 possibilities. Each time an encoded data byte is retrieved, four parity bits are calculated. The value of these bits reveal the presence of a single or double bit errror. If the error is in a single bit, the correct data can be determined from three of the parity bits; if the error is two or more bits, the computer informs the operator or attempts to retransmit the data. Other Hamming codes can be used to correct errors of two bits.

(3) Checksums. Another correction and detection technique for single or double bit errors involves the use of a quantity known as a checksum. In this method, the positional values of each "on" bit in a data sequence are added together to create the checksum, which is transmitted as part of the data. To avoid a large checksum requiring many bits, modulus 16 or modulus 2 arithmetic may be used, in which the checksum is reduced by 16 or 2 respectively, each time it exceeds these values as it is computed bit by bit for serial data transmission. When a piece of data is retrieved, the checksum is recalculated at the receiving end and checked against the transmitted value. Like Hamming codes, more checksum bits may be used to correct double or multiple bit errors of the random or burst variety. Checksums may be implemented by hardware in order to lessen the burden of the computer.

(4) Cyclic redundancy check (CRC) error. A commonly employed technique, where error detection without correction is desired, uses CRC bits. The CRC bits are generated by treating the data as a binary polynomial which is divided by a constant generator polynomial. The remainder of the division is the CRC value. Each block of data is transmitted with its CRC data and is matched to the CRC value recomputed by the receiving hardware (or computer). If the CRC values do not match, the entire block of data is retransmitted. Care must be taken to insure that the correct block of data is retransmitted; this requires a degree of cooperation (known as "protocol") between the data sending and receiving circuitry.

Section III. COMPUTER LOGIC

III-1. Basic considerations. The field of mathematics deals with the logic of binary numbers, known as Boolean alegbra, which is a special case of the more general mathematical theory of sets. A set is made up of a collection of elements or quantities capable of being represented by a number or symbol. The algebra of sets is concerned with the relationships of two or more sets of these elements and the logical conclusions which can be made about their contents.

a. Set theory. The two basic operations to be considered are the "union" and "intersection" of two or more sets, represented by their respective symbols "$\cup$" and "$\cap$". The union of sets A and B is represented pictorially by a Venn diagram, shown in Figure III-1.a, and is represented mathematically by the expression:

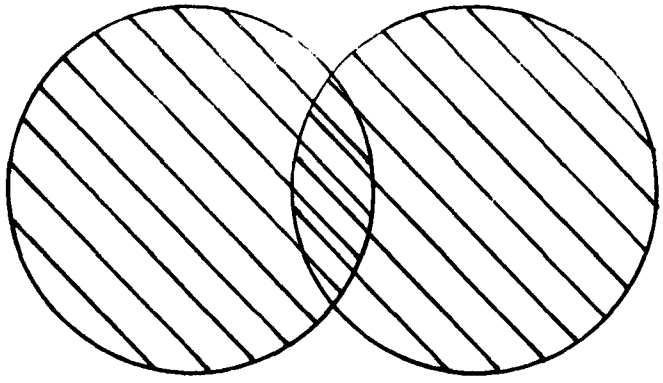
$$C = A \text{ "}\cup\text{" } B \qquad \text{(III-1)}$$

In words, this can be stated as: "The union set C consists of all of the elements of either A or B or both A and B". The intersection of sets A and B is represented in Figure III-1.b, and is represented mathematically by the expression:

$$C = A \text{ "}\cap\text{" } B \qquad \text{(III-2)}$$

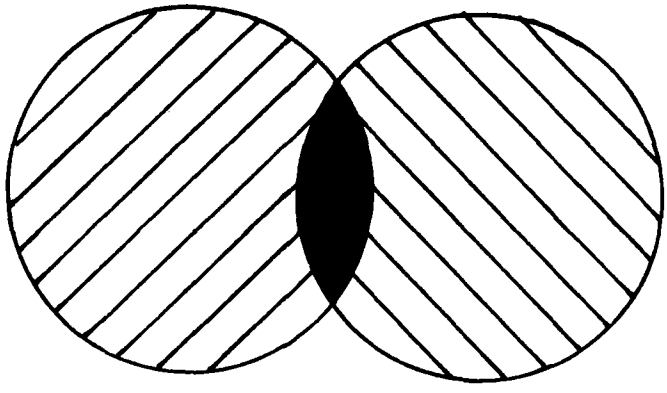
In words, this can be stated as: "The intersection set C consists of all of the elements common to A and B". These statements can be generalized to as many sets as are desired. Another basic concept of sets is the null set. The null set of A, expressed as $\bar{A}$, is the complement of set A and consists of all elements not included in A. If the set A would consist of everything outside the circle.

a)



Union

b)



Intersection

Figure III-1 Algebra of Sets: Venn Diagrams

b. Boolean algebra. In Boolean algebra, only two elements exist: 0 and 1. The union and intersection operations are referred to in Boolean Algebra as "OR" and "AND" operations, respectively. Their notation is symbolized in various texts as ".", "X", or "V" for "AND" (intersection) and "+", or " " for "OR" (union). The "NULL" operation is usually expressed by the bar over the symbol. The logic operations of Boolean algebra are summarized in rules known as Huntington's postulates, summarized in Table III-1. The symbolic logic of the Boolean algebra rules must not be confused with the operations and results of simple arithmetic. For instance, equation (2) of Table III-2., A + A = A, makes no sense in ordinary arithmetic. In Boolean algebra, its meaning is that the union of two identical inputs, both either 0 or 1, result in an output equal to whatever the input is. Equation (3) states that A + 1 = 1. The meaning of this is that the union of input A (either 0 or 1) and an input with a constant value of 1 result in an output equal to 1, since the output equals A or 1. Like the more general set theory, Venn diagrams may be used to visualize the Boolean algebra rules. However, Karnaugh maps or truth tables, as they are known, are commonly used to analyze Boolean algebra statements.

Table III-1. Boolean Algebra Rules

"OR" Functions	"AND" Functions
(1) $A + 0 = A$	(10) $A \times 1 = A$
(2) $A + A = A$	(11) $A \times A = A$
(3) $A + \underline{1} = 1$	(12) $A \times \underline{0} = 0$
(4) $A + \overline{A} = 1$	(13) $A \times \overline{A} = 0$
(5) $A + B = B + A$	(14) $AxB = BxA$
(6) $A + (B+C) = (A+B) + C$	(15) $Ax(BxC) = (AxB)xC$
(7) $A + BxC = (A+B)x(A+C)$	(16) $Ax(B+C) = (AxB)+(AxC)$
(8) $A + (AxB) = A$	(17) $A \times (A+B) = A$
(9) $\overline{(A + B)} = \overline{A}x\overline{B}$	(18) $\overline{(A \times B)} = \overline{A}+\overline{B}$
	(19) $\overline{(\overline{A})} = A$

A, B, or C may take values of 0 or 1

III-2. Truth tables. Each Boolean algebraic quantity (A, B, C, etc.) has two possible values, 0 or 1. When N quantities are used as inputs, there are 2^N possible input combinations or states. The number of output state combinations depends on the Boolean algebra operation being performed. In the simplest case of two inputs, A and B are considered with the Boolean "AND" operation. There are four possible input state combinations:

A=0	A=1	A=0	A=1
		or	
B=0	B=0	B=1	B=1

The output conditions can be determined by using Boolean algebra rules (14) and (16) from Table III-1. To illustrate this, consider the A=0, B=0 case. Equation (16) states: A x 0 = 0. Therefore, if A=0, then 0 x 0 = 0. Consequently the result of A=0 and B=0 (AxB) is 0. Note that if A=1 and B=0, AxB has the same "0" result. By symmetry, for A=0 and B=1, the result is 0. For A=1, B=1, equation (14) is used: A x 1 = A. This implies that the result is 1 when A is 1 (also true for B=1). It is convenient to arrange the above results in a truth table:

A	B	A x B Result
0	0	0
0	1	0
1	0	0
1	1	1

By the same reasoning, a Truth Table can be constructed for A + B:

A	B	A + B Result
0	0	0
0	1	1
1	0	1
1	1	1

Truth tables can be constructed for every possible Boolean algebra statement, including all of those shown in Table III-1, and extended to any number of input variables. Logic designers and manufacturers use truth tables extensively in designing or presenting their product literature.

III-3. de Morgan's Theorem. Equations (9) and (18) of Table III-1. are special cases of a very important relation in Boolean algebra known as de Morgan's Theorem. The general form of de Morgan's Theorem is

$$f(A, B, C..., +, x) = f(\overline{A}, \overline{B}, \overline{C},...,x, +) \qquad \text{(III-3)}$$

The meaning of this equation is that an equivalent form of any Boolean algebra formula involving a function of A, B, etc. and the "+" (OR) and "X " (AND) operators may be derived by interchanging the operators and inverting the set polarities (A to $\overline{A}$ or $\overline{A}$ to A). This general form of the theorem is stated without proof; however equation (12) is shown to be true by means of the truth tables shown in Table III-2. The usefulness of de Morgen's Theorem lies in the fact that many operators can be eliminated from any logic statement, thereby enabling a computer designer to construct the logic circuitry of a computer out of fewer different kinds of components resulting in considerable cost savings. This will become more apparent in the next paragraph.

Table III-2. Proof of equation (12) of Table III-1.

a) A+B

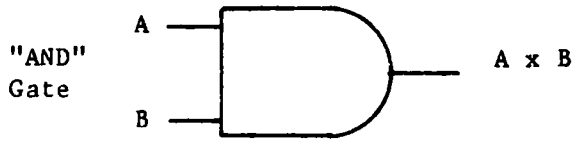
INPUT		OUTPUT	
A	B	A+B	$\overline{A+B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

b) A x B

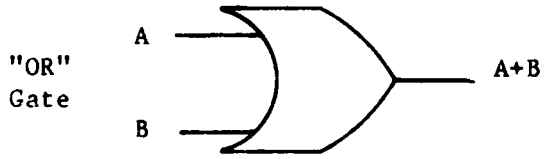
INPUTS				OUTPUT
A	B	$\overline{A}$	$\overline{B}$	$\overline{A}x\overline{B}$
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

III-4. Basic digital logic.

a. Standard symbols. Standard symbols are used to indicate electrical circuits performing Boolean algebra operator functions. The symbol for the "AND" operation for two inputs, known as an AND gate is given by:

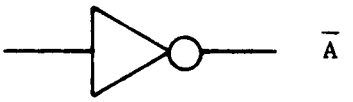


The symbol for the "OR" operation is given by:



The third basic operation of Boolean algebra is the "NULL", which indicates the complement of the input and output value. Electrically, a device called an inverter is needed to perform this function symbolized as:

"inverter" gate



The AND, OR and inverter (NULL) functions or gates are contained on semi-conductor chips, or integrated circuits (IC's) that contain a number of transistors, resistors and diodes capable of performing logic operations. The IC,s are categorized nominally into three groups to distinguish the physical density of gates per chip they contain: small scale integration (SSI) devices contain 12 or less gates; medium scale integration (MSI) devices contain more than 12 but less than 100 gates; and large scale integration (LSI) devices contain more than 100 gates, generally in the thousands. Rapid advances in technology promise the introduction of very large scale integration (VLSI) in the immediate future, with gate densities on the order of tens of thousands. SSI devices were the first IC's to appear, but they are by no means obsolete. Several major generations of SSI technologies have evolved, the major ones being:

(1) Diode-Transistor Logic (DTL). DTL uses directional variation of diode resistance to produce the switching effect. In this type of circuitry, transistors are used as unit gain amplifiers to enable the inputs and outputs of the diodes to be connected to other devices without the signal degradation that is brought about by loading effects. The number of inputs a gate can accept is termed the fan-in factor; the number of gates a gate can drive is the fan-out factor. Figure III-2.(a) shows a DTL NAND gate. DTL IC's are still used to some extent but the technology is no longer under development and has been superceded by transitor-transistor logic (TTL) technology devices. ces.

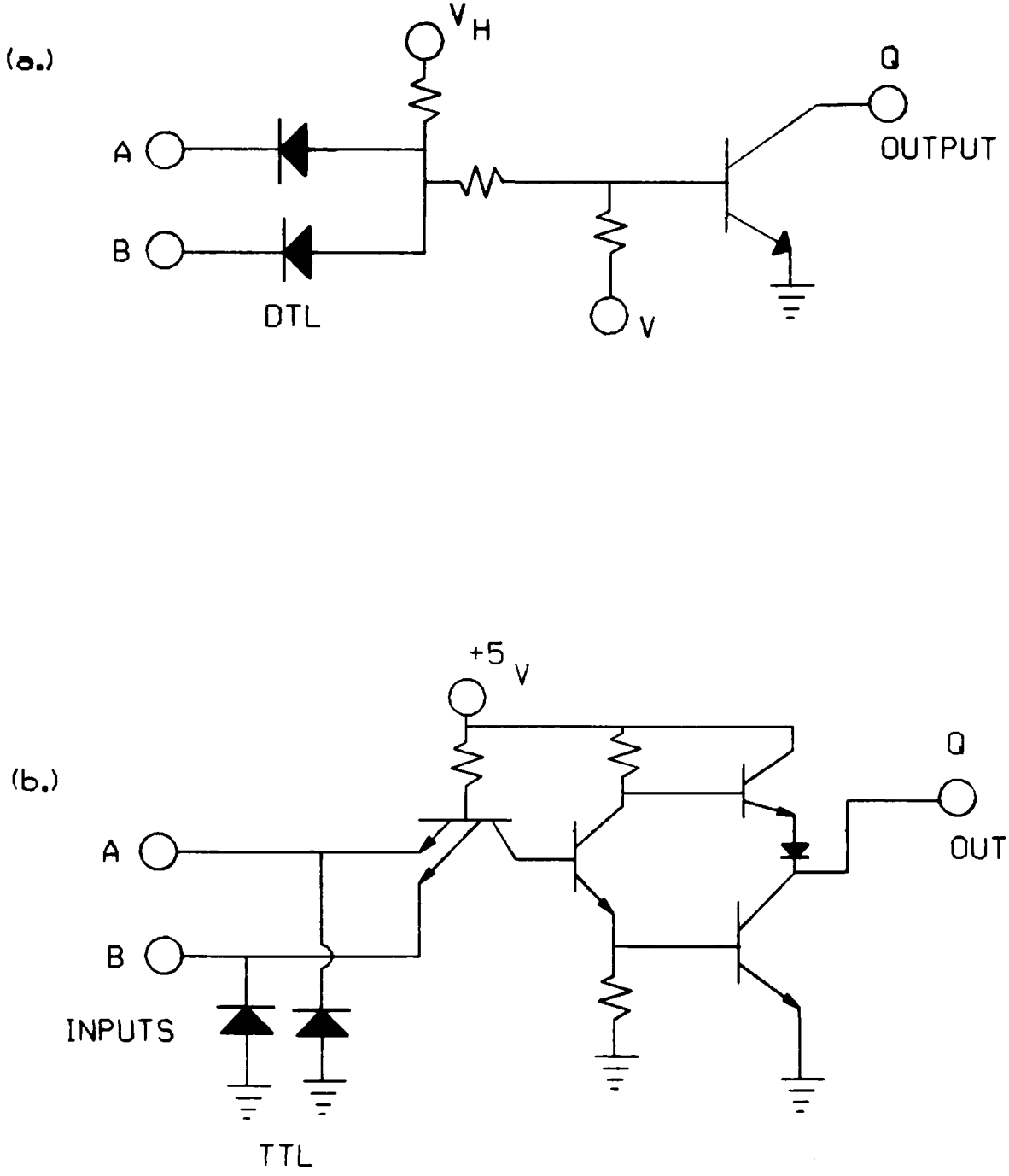


FIGURE III-2. DTL AND TTL NAND GATES.

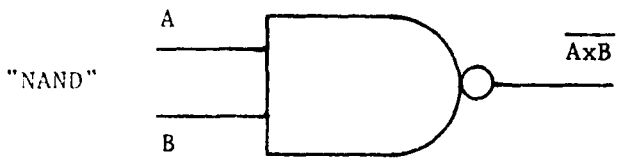
(2) Resistor-Transistor Logic (RTL). RTL is a largely obsolete logic technology using resistors and transistors. It is a peculiar fact of semiconductor IC technology that resistors are more difficult and costly to fabricate than transistors, a reverse of the situtation encountered with discrete electronic components.

(3) Transistor - Transistor Logic (TTL). TTL is currently the most popular technology for SSI devices and is characterized by the replacement of diodes with transistors having multiple emitters, as illustrated by the TTL gate in Figure III-2.(b). More importantly, the TTL devices have a specified common level of voltages. A logic state of "1" (or high state) is interpreted when a level of 2-6 volts is sensed. Generally, 5 volts is specified for the high state. The logic state of "0" (or low) binary state is interpreted as being below .2 volts. Devices are said to be TTL compatible if they observe these conventions. Some TTL devices are supplied without an internal power supply resistor, known as the "pull up resistor", thus permitting the parallel hookup of the inputs or outputs of several devices whose signals may be effectively connected in a "Wired-OR" logic circuit.

b. MSI and LSI Devices. MSI and LSI devices contain a number of gates, generally for special applications. A microprocessor is one such LSI device, and will be discussed in detail in later Sections.

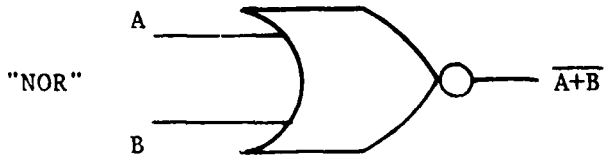
c. NAND and NOR Functions. It is convenient to combine the AND and INVERTER and the OR and INVERTER functions on a single IC device to make a variety of other logic operations available. The most useful of these are the NAND and NOR gates.

(1) NAND Gate. The NAND gate gets its name from a contraction of "NULL AND", of which it is an equivalent function. Symbolically, the NAND gate is shown as an AND gate with the addition of a dot on the output to indicate inversion, as shown with an accompanying truth table:



A	B	AxB	$\overline{AxB}$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

(2) NOR Gate. The NOR gate gets its name from a contraction of "NULL OR". Like NAND, the NOR gate is shown as an OR gate with the addition of a dot on the output to indicate inversion. A NOR gate is shown as:

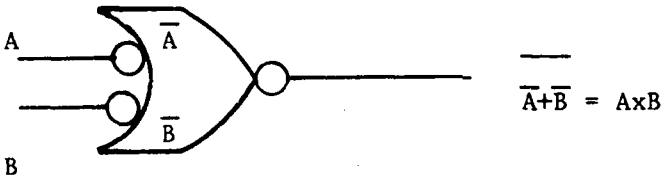


The Truth Table for NOR is the same as that shown in Table III-2.a.

(3) Negative and positive logic devices. A study of de Morgan's Theorem indicates that an AND function can be performed with a NOR gate and INVERTERS on the inputs.

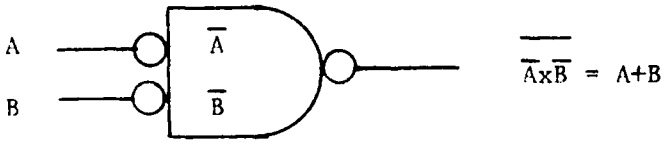
$$A \times B = \overline{\overline{A} + \overline{B}}$$

The term $\overline{\overline{A} + \overline{B}}$ represents a gate with the following symbol and truth table:



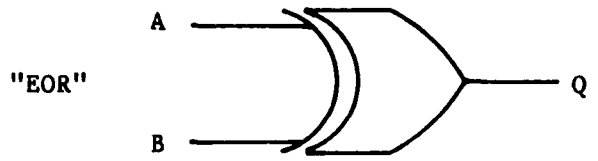
A	B	$\overline{A}$	$\overline{B}$	$\overline{A}+\overline{B}$	$\overline{\overline{A}+\overline{B}}$
0	0	1	1	1	0
0	1	1	0	1	0
1	0	0	1	1	0
1	1	0	0	0	1

As can be seen, the last column of the above truth table is identical to the truth table for AND shown previously. Likewise, it can be shown that an OR function can be expressed by a NAND device with INVERTERS on the inputs:



The utility of these equivalencies lies in the fact that NAND and NOR gates, sometimes referred to as negative logic devices, consume less operating power, than positive logic devices (AND, OR gates), an important consideration to designers and manufacturers. Thus, most TTL IC's are NAND and NOR gates packaged with or without input INVERTERS.

d. EXCLUSIVE - OR function (EOR). Another logic function which has proven useful is the Exclusive-OR (EOR) function. The EOR gate is shown symbolically as



The EOR function is similar to the OR function with the exception that when inputs A and B are both "1", the output Q is "0" instead of "1". Thus, the truth table for EOR is

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

The EOR function is useful in performing the complementing of binary quantities in arithmetic operations. It replaces several separate AND, OR and inverter gates, since the simplest form of the EOR function in terms of AND and OR gates can be shown to be mathematically equal to:

$$Q = (A \times \overline{B}) + (\overline{A} \times B)$$

III-5. Integrated circuits. Using combinations of the logic gates described in the preceding paragraph as building blocks, logic circuits with any degree of complexity may be constructed. The gates described were shown with two inputs; however, gates with three or more inputs are available. By sometimes viewing the binary 0 (or "low") state as "true" in a truth table, it is possible to design computers entirely with combinations of NAND gates and inverters to minimize costs.

Section IV. DIGITAL COMPUTER ARCHITECTURE

IV-1. Definition and major requirements of a computer.

a. Six criteria for computers. The theoretical concept of the computer evolved slowly throughout the history of mathematics. The computer represented the practical desire of mathematicians and others to eliminate the tedious manual operations of arithmetic used in computing mathematical tables such as logarithms, astronomical tables, military ballistics, etc. The first five important criteria defining the necessary characteristics of a practical computer were first set down by Charles Babbage in 1830. An additional criteria was proposed by Von Neumann in 1947. These criteria state:

(1) An input means must exist for entering data in the form of numbers to be manipulated (operands) or instructions (operators) to direct the operation of the machine.

(2) A storage system must exist to hold both the operands before and after they are operated upon, and also the operating instructions.

(3) An arithmetic calculation section must exist that can perform the instructions and retrieve or store the operands.

(4) A medium must exist for delivering the results (output) of the computations to the user.

(5) The machine must be capable of deciding between alternative sequences of operations depending upon the value or state of variables.

(6) Von Neumann proposed that the numerical data and machine instructions should be stored in the same format, indistinguishable to the machine. This would allow the machine to modify its own instructions. This last criteria has proved to be a very powerful concept, enabling the computer to operate faster and more efficiently, as well as reducing many of the programming difficulties.

b. Practical computers. The first practical computer based on the above criteria was completed in 1944. The machine utilized electro-mechanical relays and switches, thus severely limiting its operating speed. Shortly after this, work began on a computer utilizing vacuum tubes to act in place of the relays and switches, a machine capable of far faster operation. The first commercially produced computer appeared in 1951, utilizing the vacuum tube technology. At this time, the newly invented transistor was making its presence felt. Despite some disadvantages such as lack of ruggedness and sensitivity to temperature extremes, the transistor has superiority over the vacuum tubes for computer applications because of the following reasons:

(1) Lower power consumption; no filament power required.

(2) Physically smaller size for equivalent circuits.

(3) Cheaper to produce in large quantities.

(4) More reliable operation; more hours between equipment failure.

c. Deficiencies of early computers. The last criteria, reliability, is an important consideration. The early vacuum tube computers where massive, power hungry behemoths in constant need of maintenance. The transistor helped alleviate that problem. Another major defect in the early computers was the necessity of programmers to work solely in binary machine language. Writing a program was a formidable task requiring a great deal of time and tedious manual effort in entering the program into the machine. Mistakes were inevitable, and thus, only very important problems could justify the expense of computer solution. The introduction of assemblers and finally, high level languages such as FORTRAN in the late fifties made the computer a practical device available to a wide spectrum of the public.

d. Organization of computer equipment. Although computer systems vary greatly both internally and externally and use a wide variety of electro-mechanical devices, the major components perform according to the criteria requirements as set down by Babbage and Von Neumann. The computer equipment ("hardware") and internal operating system ("software") architecture must encompass the components depicted in Figure IV-1. By convention, the computer control and processing arithmetic-logic unit have together come to be known as the central processing unit (CPU). Closely associated with the CPU is the central memory, frequently referred to as core, because of the widespread use of magnetic cores as memory storage devices. The input and output components are usually referred to as peripheral devices. They include such hardware as cardreaders, magnetic tape drives, magnetic disks, cathode ray tube (CRT) terminals, keyboards, and printers.

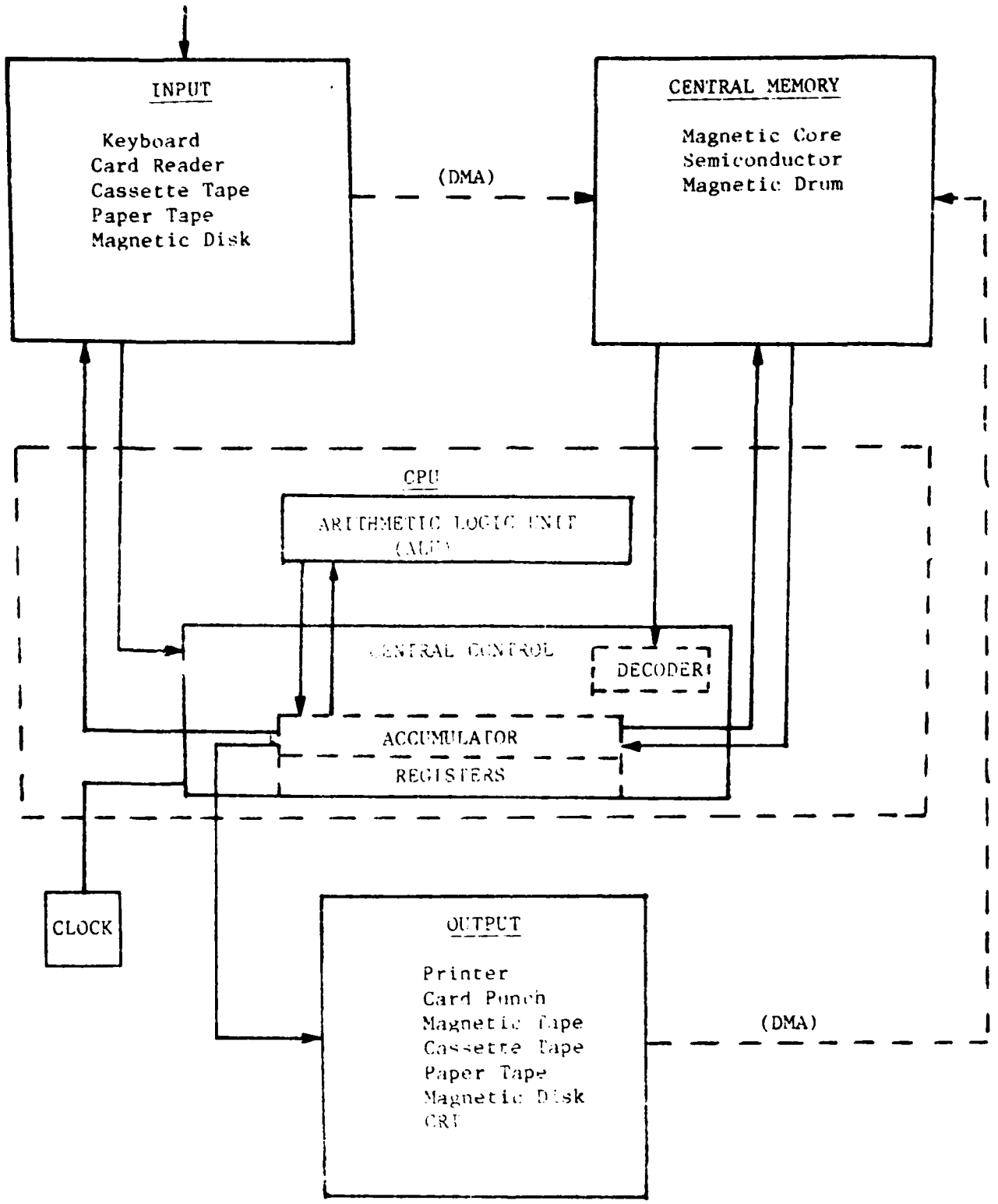


Figure IV-1 Essential Components of a Computer

IV-2. Computer memory storage. In presenting details of computer architecture, it is useful to begin with a discussion of the memory system because of the bearing that memory has on the architecture of the CPU. The memory of a computer is organized around the characteristic data group, or word, by using parallel lines to each bit location. The common lines on which data travels to and from memory are called the data bus. The common lines which select a particular word in memory are called the address bus. Modern general purpose mini and mainframe digital computers have word sizes that commonly range from 12 to 64 bits. Word lengths of 8 bits or less appeared with the first generation of microprocessors, but 16 bit word length machines are now available, with longer word lengths expected. The word length is directly related to the speed and accuracy with which the computer performs arithmetic. In order for the CPU to be able to access a particular word of memory, each word must be assigned a unique address. The number of words accessible to the CPU by direct addressing is limited by the number of lines on the address bus. For instance, a CPU with 16 address lines can access a maximum of 216 or 65,536 (expressed as 65 k) memory locations. In actual physical terms, there are several methods and media for storing data currently in use, each having its own characteristics of speed and capacity. Many computer systems use a variety of these storage media. An ideal property of one type of computer memory is that the time required to access every word in memory is the same because each storage location is accessible by X and Y coordinates. Memory with this characteristic is known as random access memory (RAM).

a. Flip-flops. The first practical random access memory devices consisted of flip-flops, which utilized a pair of vacuum tubes configured in a bistable arrangement. In this circuit arrangement, positive feedback is used to hold a circuit element such as a vacuum tube in a conducting or nonconducting (on or off) state after an input signal is removed. Vacuum tubes proved to be unreliable and flip-flops composed of them had the added disadvantage of volatility, that is, interruption of power caused loss of memory information.

b. Magnetic Drum. The magnetic drum was the next storage medium to be introduced. This device is a cylindrical component with an outer surface covered with a magnetizable medium. Binary information is detected by a moving magnetic head to determine the absence or presence of a magnetic field under the rotating drum. The magnetic drum rotational speeds put a limitation on the rate at which the CPU is able to retrieve or store data. Even at 9600 rpm, retrieval of a word of data could take as much as 625 miliseconds, a time period much longer than the cycle time between the CPU instructions of a typical computer. Thus, magnetic drum devices (or the magnetic disk to be described) are unsuitable for use as main memory for the CPU. They are, however, well suited for mass storage of data which can be emptied or dumped from faster central memory storage systems such as magnetic cores. Magnetic drums are presently utilized for mass storage by only one remaining manufacturer. The chief advantage of drum versus other mass storage devices are physical ruggedness, and higher data transfer rates. However, data recording densisties are generally lower. Magnetic drum and disk devices are

not truly random access devices, since the read or write times are dependent upon the location of the magnetic head relative to the position of data on the rotating drum.

c. Magnetic core. Magnetic core storage utilizes the direction of the magnetic field in a highly permeable toroidal element to indicate the presence of a binary 0 to 1. To sense the polarity of the magnetic field on each bit of the word, special circuitry is required. The core bit arrays are supported by wires passing through them. These wires include a write winding, a sense winding to detect magnetic polarity, and a clear winding to zero the bit. Each time the core is "read", the information that is stored is destroyed and thus must be rewritten into the memory. The advantages of magnetic core memory are high speed input and output (I/O), low cost per bit, and non-volatility. Disadvantages of magnetic core memory are high power consumption, physical bulkiness of the core arrays and power supply, and complicated, difficult to maintain circuity. Magnetic core memory is still the most popular central storage medium for the large scale computer systems known as mainframes.

(4) Semiconductor memories. The recent advent of LSI IC technology has led to the widespread use of semiconductor RAM. Most of these devices are based on the simple flip-flop circuits which were discussed above. A simple semiconductor flip-flop circuit built out of two NOR gates is shown in Figure IV-2.a. Its input timing diagram is shown in Figure IV-3. When both the set and reset inputs are at 0, the output Q is 0. This output and the RESET, which equals 0, becomes the inputs to the second NOR gate. The output of the second NOR gate is then 1, and it is fed back to the second input of the first NOR gate. Thus, even when the SET=0 pulse is removed, the output of the first NOR gate, and the flip-flop as a whole, remains at 1. Only by bringing the RESET to 1 is the flip-flop output brought to 0. The SET and RESET inputs are never brought to 1 at the same time. Semiconductor RAM IC's consist of as few as 8 flip-flops to over 64 thousand on a single package (chip); each flip-flop contains 1 bit of data. Generally, the RAM flip-flops operate under clock control; that is, they can be activated to a 0 to 1 state only after a timing pulse is sent by the CPU. Several types of clock controlled flip-flops exist:

(a) RS Flip Flop. The R (Reset) or S (Set) inputs of the RS flip-flop, shown in Figure B-5, must be 1 during the leading edge of the clock timing pulse to change the state of the output. Two auxiliary inputs are included, a PRESET and CLEAR, neither of which are under clock control. Both of these inputs are normally at the 1 level. Grounding the PRESET forces the flip-flop into the 1 state; grounding the CLEAR forces the flip-flop into the 0 state.

(b) RS master-slave flip-flop. This flip-flop consists of two RS flip-flop in series. The first flip-flop transfers incoming data to the second flip-flop, which operates on the trailing edge of the clock pulse; thus the output timing is offset by the width of a clock pulse.

(c) JK flip-flop. This flip-flop is similar to the RS flip-flop with the exception that when both the J (SET) and K (RESET) inputs are 1, the clock

pulse will complement the output to the opposite of whatever state existed. A JK master-slave flip-flop is a flip-flop pair analogous to the RS master-slave flip-flop.

(d) D flip-flop. The D flip-flop contains only one clocked input line (D) which determines the output state. The leading edge of the clock pulse activates the flip-flop.

(e) Latch. A Latch is a special flip-flop which, like the D flip-flop has only one input. It is used to sample data when the changing data interval is shorter than the clock. The output of the latch is determined by the input state which exists at the trailing edge of the clock pulse.

(f) IC and RAM manufacturing technologies. Semiconductors are manufactured by utilizing a variety of high technology processes and produce devices with a wide range of characteristics and performance. A common factor in the production of semiconductors is the use of highly refined, single crystal silicon cut into thin wafers. Impurities are carefully and selectively introduced in the silicon wafers by a process called doping, to produce the semiconductor effect. Originally, only discrete devices such as diodes or transistors were produced by these processes. Later, it was discovered that through the use of microscopic "masks", a number of resistors diodes and transistors could be etched onto multiple cells of a single small silicon wafer (or "chip"). This process was first applied to bipolar transistor technology to produce IC gates featuring very high speed but relatively low gate density. More recently, a technology based on metal oxide semiconductors (MOS) has achieved a high degree of popularity. This technology is characterized by the use of etched field effect transistors (FET) controlled by the electric field around a doped MOS gate. Several different types of MOS devices exist which use P-type (PMOS) (for positive charge), N-type (NMOS) (for negative charge) semiconductor material, or a combination of the two called complementary MOS (CMOS). Each type of MOS device is characterized by differing speeds, gate density, and fabrication cost. The latest development in MOS, a CMOS process grown on a substrate of sapphire material, known as silicon-on-sapphire (SOS) technology, exhibits very high speed capabilities. Low power consumption is a desirable characteristic in semiconductor devices. To achieve this, the Schottky diode clamp has been applied to TTL logic gates, leading to similar technologies for low power devices. Another recent technology, integrated injection logic (IZL) has the potential to become popular in the near future because it exhibits the traits of high speed (near that of bipolar), a greater gate density than MOS devices, and low power consumption. These technologies are used to manufacture not only RAM's but other commonly used logic IC's such as TTL. RAM's are commonly produced with a range of operating cycle times. Higher speed devices are more expensive; thus RAM's are selected to match operating speed of the CPU's they serve.

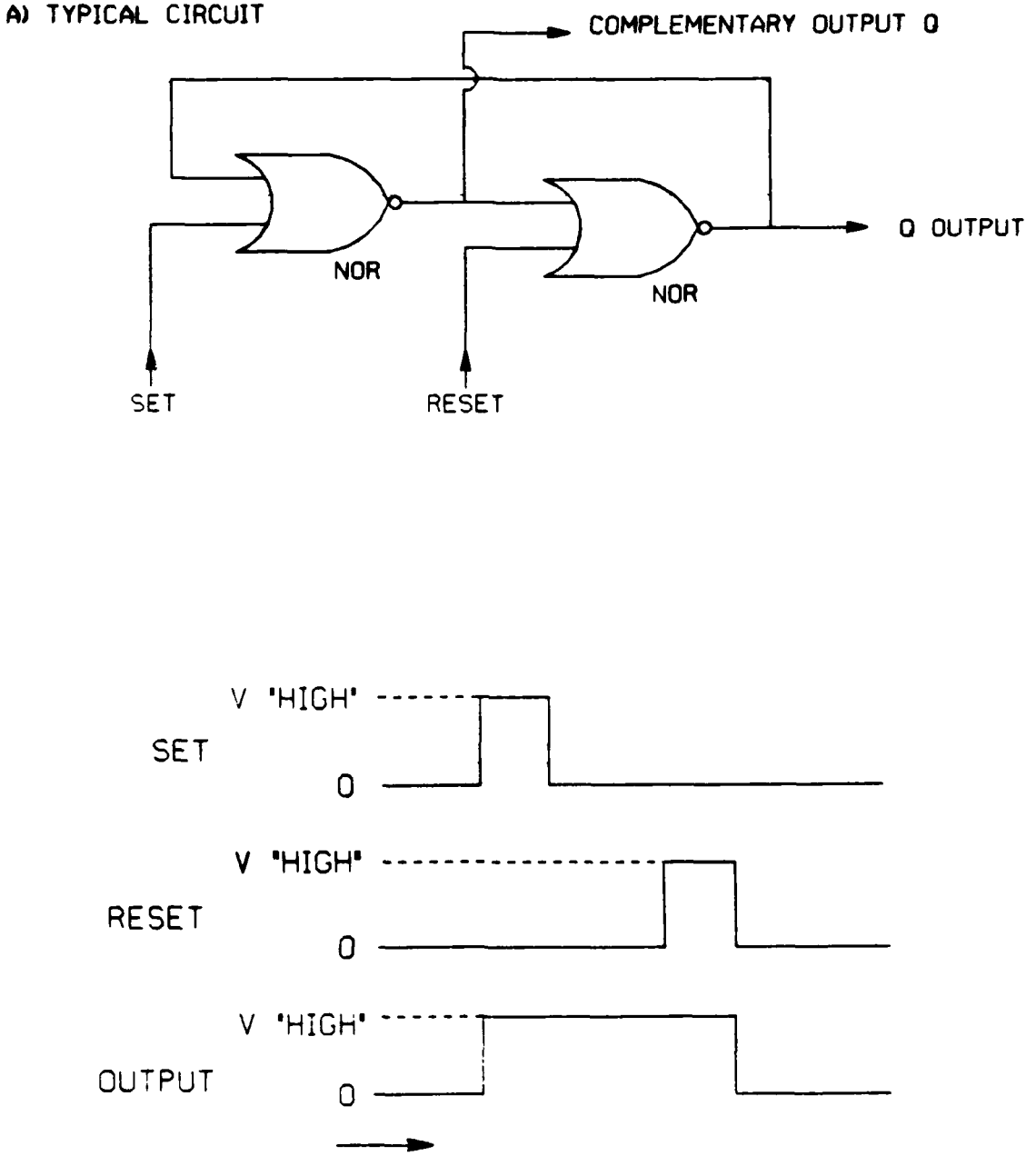


FIGURE IV-2. FLIP-FLOP

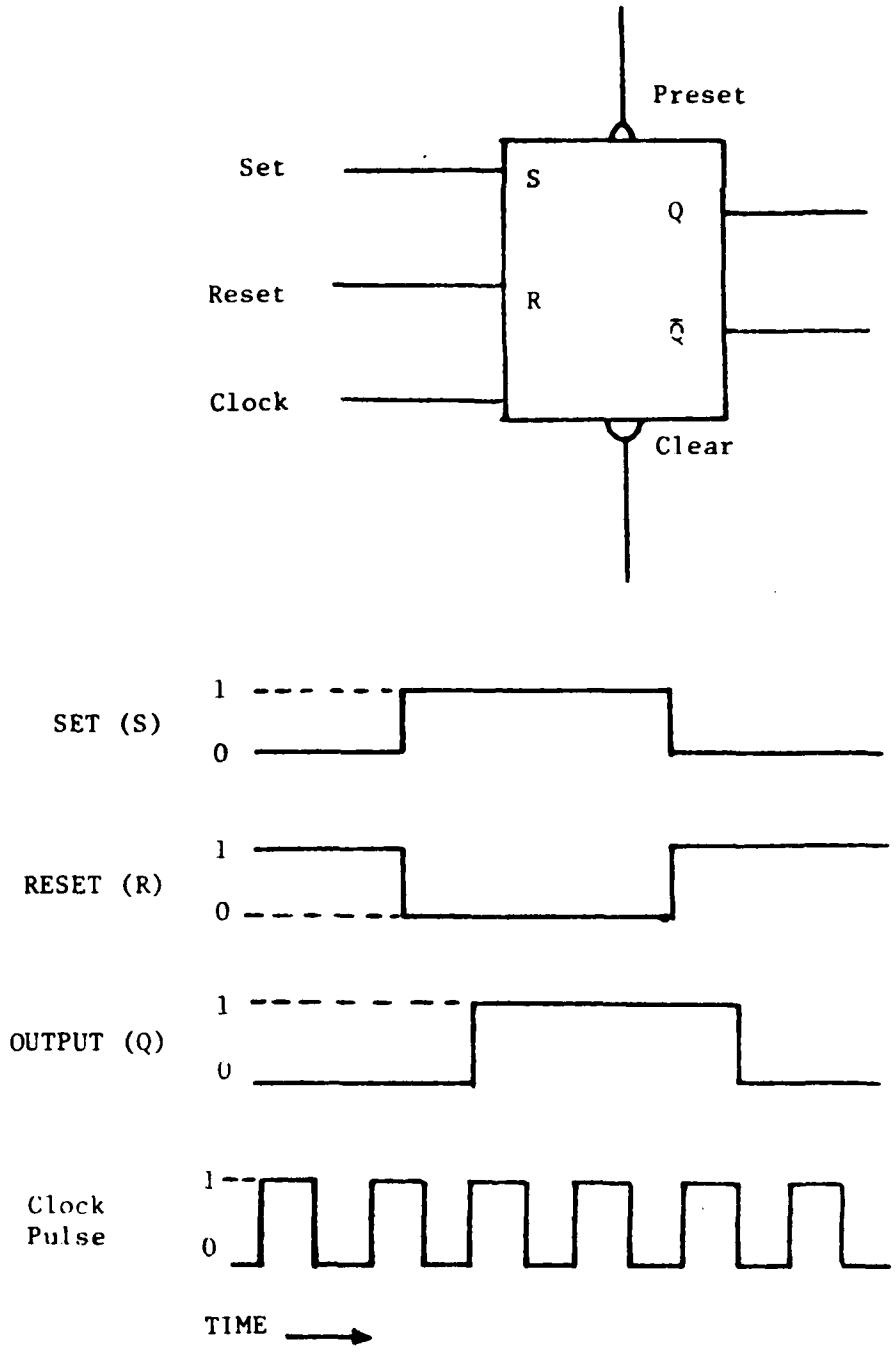


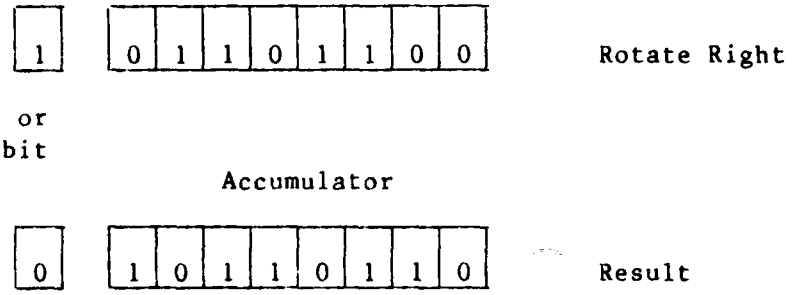
Figure IV-3. Timing of an RS Flip Flop

IV-3. Central processing unit (CPU).

a. Architecture. The organizational structure of a digital computer system, known as architecture, varies greatly according to the intent and design philosophy of the various manufacturers. However, a number of common design features are evident. The computer CPU is made up of a small number of memory storage locations called registers, each typically containing a word of data. In addition, the CPU contains logic circuitry the purpose of which is to interpret machine instructions and provide paths for data to travel between registers, or between the registers and memory storage areas outside the CPU that are directly accessed by using the address and data bus as previously described.

b. Registers. The number of registers provided in a CPU is determined by the computer designer and manufacturer. The registers are made up of a series of flip-flops containing the characteristic number of bits in a word. Several specialized registers are nearly always included.

(1) Accumulator. The accumulator is the most important register in the CPU because it provides the "scratch pad" location in which all arithmetic and logical operations take place for data travelling to and from the main computer memory storage. The accumulator consists of a number of bits equal to one computer word plus an extra bit in the most significant digit position known as the carry (or "link") bit. The carry bit is used to indicate when the result of an arithmetic operation exceeds the capacity of the accumulator. Another function of the carry bit is its use as a "flag" bit. This function is connected to a machine instruction which shifts or rotates the data in the accumulator in a loop right or left as shown below for an eight bit accumulator:



Through the use of rotation instructions and of the carry bit, the state of individual bits in a data word can be tested to choose between alternate actions. The computer can perform arithmetic and logical operations with only two numbers (operands) at a time. One operand is held in the accumulator while the other usually is obtained from other registers or from memory storage. The contents of the accumulator may or may not be altered depending upon the type of instruction encountered in the program.

(2) Program Counter (PC). The PC is a register whose purpose is to keep track of the address of the instruction to be performed. The CPU operates by putting the value stored in the PC on the address bus. The data stored in the specified address travels to the CPU via the data bus where it enters the instruction decoding circuits. While this is going on, the PC is automatically incrementd by 1 so that it will "point" to the next instruction. Generally, the computer will perform the instructions stored in memory in a sequented manner. However, at times it is often necessary to skip to another portion of memory by a process known as branching. When this happens, the PC is modified to a new value by the CPU circuits, after which it again begins stepping sequentially through the memory. The PC frequently contains more bit positions than the word length of the computer to enable it to address more memory locations. The number of bit positions in the PC is usually equal to the number of address lines. Theoretically, an accumulator and PC are the only registers necessary to allow a computer to operate, although in an inflexible manner.

(3) Status register. The CPU contains a number of individual flag bits to indicate several conditional states which can be tested under program control if so desired. These independent individual bits can be considered to be single status register. The conditions a CPU commonly monitors in the flag bits include but are not limited to the following:

(a) Zero result flag. The result of the CPU operation leaves zero in the Accumulator. Under this condition the zero flag is set.

(b) Negative flag. The result of the CPU operation leaves a negative number in the accumulator. Under this condition, the negative flag is set. When the CPU uses two's complement arithmetic, this also means that the most significant bit in the accumulator (not including the carry bit) is set. The usefulness of this concept will be explained in the paragraph dealing with branching instructions.

(c) Overflow flag. If, after any arithmetic operation, the resulting operand has a greater magnitude than can be expressed by the bits in the accumulator, the overflow bit is set. The importance of this bit to the programmer depends upon whether or not signed arithmetic is being utilized. When the arithmetic is unsigned, the overflow bit is ignored. In signed arithmetic, the overflow bit has the same meaning as the carry bit to the programmer and indicates that a sign correction routine must be used when this bit is set after an addition or substraction instruction.

(d) Interrupt disable flag. The usefulness of this flag bit will become clearer following the discussion of interrupts in a later paragraph. Briefly, it inhibits the ability of a signal from an outside device connected to the computer to interrupt the programmed operation of the CPU.

(e) Carry bit flag. The carry bit has already been discussed. All of the above flag bits may be set or reset by the programmer and may be used to make logical decisions under program control by the CPU.

(4) Stack pointer (SP). The concept of the stack is a fairly recent one in the development of computers and represents an added dimension of power for the programmer. Its use will become clearer during the discussion on machine instruction sets. The stack, sometimes called a push-down stack, is a special area set aside in memory used for the storage of data from the CPU registers. The data is stored sequentially on a last-in, first-out (LIFO) basis. The SP initially points to the highest numerical address in the stack. Each time data is "pushed" onto the stack, the SP is decremented by one to point to the next lower available address. Data may be retrieved or "popped" from the stack by a reverse process in which the SP is incremented by one each time. Not every CPU utilizes stack architecture in this described manner. Some stacks are actually special registers in the CPU while others use memory outside the CPU as described. The number of stack locations is determined by the computer designer, and the programmer must be aware of these limitations.

(5) Index registers. Many CPU's are provided with varying numbers of index registers. These registers are commonly used as indexes or counters to free the accumulator for more important tasks. The usual method of using an index register is to load it with an integer value which is then decremented by one until it reaches zero, a common programming operation utilized in repetitive loops and timers. Index registers are also used by the programmer as intermediate storage areas for arithmetic and logical operations.

(6) Other registers. Some CPU's are provided with additional general purpose registers available to the programmer. Their chief benefit is that of conserving operating time, since data transfers between registers are commonly faster than transfers between the accumulator and memory. A second accumulator is provided in some CPU's to allow additional latitude and operating speed.

c. CPU clock. All semiconductor devices require a finite time to pass the input data through to the output because of the limitations of the speed of light in a material medium. This time, known as cycle time, varies from device to device. In the CPU, some logic devices must "wait" for others while the data signals stabilize in order to avoid ambiguous data conditions. The operation of the CPU is then limited to that of its slowest logic circuit components. To synchronize the operation of the CPU, a clock is utilized. This clock outputs a continuous stream of square wave pulses at an unvarying frequency. On the positive peaks, the individual logic circuits and the RAM's are activated or enabled simultaneouly. Data is read, written or otherwise manipulated only during the "on" pulses of the clock, although data may be in transit in between clock pulses. In a typical computer system the address bus will change during the first half of a clock cycle and the data will be transferred in the latter half. Some CPU systems utilize multiphase clocks that derive their pulses from a single timing source.

d. Arithmetic logic unit (ALU). The ALU portion of the CPU performs the arithmetic or logical calculations in a computer. Generally, an ALU is capable only of performing addition, AND, OR and EOR functions. Some recent CPU ALU's are designed to perform multiply and divide functions and floating point arithmetic operations as well. Normally, however, functions more complicated than integer addition are handled by software programs supplied with the computer.

e. Control unit. The control unit of the CPU contains circuitry to decode program instruction codes, increment or decrement registers and act as a gate for data paths between all parts of the computer. In order to maximize the operating speed of the CPU, most control units are capable of performing more than one operation at a time by utilizing a technique called pipelining. As an illustration, while a CPU control unit is sending address information to retrieve ("fetch") data, it may be simultaneously decoding the next program instruction to be executed.

IV-4. Instruction sets. Theoretically, it has been shown that a computer needs only two basic instructions in order to operate; on a practical level, however, a variety of instructions are available to the programmer. A large mainframe computer may have no more than 30 instructions, but what the CPU lacks in versatility, it more than makes up in operating speed. Minicomputers or microprocessors are usually equipped with many more instructions and a variety of addressing modes to compensate for their slower CPU operating speeds. The Zilog Z-80, an 8-bit microprocessor and the DEC LSI-11, a 16-bit microprocessor, each have well over 100 basic operating instructions in their repertoire. The various addressing modes bring the LSI-11 instruction set to over 400. Program instructions must be entered into the computer in the form of binary codes. Unfortunately, no two computers are coded in the same way. Some general features are common, however.

a. Instruction format. Instructions are written by the programmer and stored in the form of an operating code (or OP code) followed by one or more operands which usually represent memory addresses but can also represent actual numbers. The number of operands appearing after the OP code is a function of the word size of the computer. To represent 256 OP codes, for example, 8 bits are needed. If the computer word length is 32 bits, then 24 bits are available for the operand(s). In a typical microprocessor having an 8-bit word, no bits are left for the operand. In this case, the instructions are obtained by the CPU by multiple fetches, in which the OP code is first brought into the control unit to be decoded while the next sequential memory location containing the operand is being fetched. Some instructions require no operands. Most microprocessors with 8-bit word lengths contain an addressing capability of 16 bits, so that instructions requiring the absolute address of memory data cause the CPU to make two 8-bit fetches after the OP code fetch. Thus, the operation of a microprocessor is slower than that of a longer word length machine capable of fetching an OP code and operand(s) at the same time.

b. Typical instructions. It is convenient to be able to transfer data from one memory location to another. Most CPU's do not do this directly but rather have a pair of instructions to transfer data to and from a memory location via the accumulator. Many computers utilize several addressing modes to accomplish data transfer to the accumulator:

(1) Absolute addressing. The location of the operand is specified by the actual absolute memory address of the data. This mode of addressing requires the most information and is thereby slower than some other addressing modes.

(2) Relative addressing. The location of the desired data is specified as an offset to some "base" memory address, usually the address of the current instruction itself. In effect, the control unit adds the offset to the value in the program counter and puts the result on the address bus. The value of the offset is limited to one byte or one machine word of data, depending upon the particular architecture of the CPU. In the case of an 8-bit microprocessor, the offset is contained in 8 bits; thus the offset may be either a maximum of 255 words ahead of the current location or between +127 and -128 words of the current locations depending upon particular machine design. Some CPU designs have another form of relative addressing called page relative addressing. A "page" of memory consists of all the locations capable of being addressed by one memory word. A CPU with an 8 bit word has a page size of 256 memory locations; the total addressable memory (assuming 16 address lines) thus contains 256 pages of memory locations. In page relative addressing, the offset is added to the lower boundary address of a particular page, which in many CPU's is the lowest or "zero" page. Some CPU's use page relative addressing from any memory page.

(3) Indexed addressing. This form of addressing is especially useful when large quantities of data arranged in sequence is to be operated upon. The operand becomes a base address and is added to the value contained in an index register. The result is put onto the address bus and the data is fetched to be operated upon by the instruction indicated by the OP code. This process is shown symbolically by the example below in which the program instruction located at address 0100 requests the CPU to load the accumulator (expressed by the mnemonic LDA) with data located at address 1000 plus the value stored in an index register X:

Location	Instruction	Operand
0100	LDA	1000 + X

(4) Indirect addressing. In indirect addressing, the operand following the OP code acts as a pointer to another memory location. The operand is treated as an address whose contents are the actual address of the desired data. The method of implementation of this type of addressing varies greatly among machines. It is frequently combined with indexed operations in a mode called indirect indexed addressing. This mode of addressing is a form of branching, since the contents of the memory at the address designated by the operand can be altered during successive passes through the program. An illustration of indirect addressing follows:

Location Address	Memory Contents Or Instruction Mnemonic	
.		
.		
.		
0100	LDI	
0101	0200	
0102	ADD	Accumulator Contents
.	.	at beginning of
.	.	instruction 0102 = 50
.	.	
110	50	
.	.	
.	.	
.	.	
0200	110	

In the above illustration, the LDI instruction mnemonic stands for load accumulator, indirect mode, and is actually stored in memory as a binary OP code. The next address, 0101, contains the operand, 0200. The CPU control unit decodes the LDI instruction and fetches the contents of 0200 (110) and puts it on the address bus. The contents of 110, 50 in this case, are consequently loaded into the accumulator and the CPU program counter advances to the next instruction at address 0102.

(5) Immediate addressing. Immediate addressing is a special mode in which data is not actually addressed. Instead, the operand itself is the data upon which the instruction operates.

c. Instruction types. The instructions available to the programmer of a CPU may be grouped according to their basic functions. For the purposes of this manual, instruction mnemonics will be designated in groups which may or may not correspond to those encountered in actual practice:

(1) Arithmetic, logical or move instructions. Arithmetic and logical instructions allow the programmer to add (subtract) or perform logical AND or OR functions with values contained in the accumulator and the value of a particular memory location. Memory contents are not altered except when the CPU writes into them. Move instructions enable the programmer to fetch data from memory and put it in the accumulator or other registers, and conversely, to place data from the registers into memory. Logic operations are carried out on a bit by bit basis. For instance, to perform the AND operation on data contained in address 100 with binary 10101101 in the accumulator, the following steps are performed:

Operation:	Location	Binary Contents
AND 100	100	00001101

Accumulator Contents 10101101 before execution AND operation

Results in Accumulator Contents 00001101

It is seen in the above illustration that the last four digits of the number in the Accumulator are unchanged while the contents of the first four digits are altered. This process, known as masking, may be generalized to allow the programmer to use the AND instruction to selectively examine or alter bits in any data byte or word for later use. The OR or EOR instructions may be used in a similar fashion. Some possible instructions in this group include:

(a) LDA M - Load accumulator with data from memory location M. This instruction and all others affect the status of some or all flag bits.

(b) STA M - Store data from accumulator in memory location M.

(c) TAX - Transfer accumulator contents to register X.

(d) TXA - Transfer register X contents to accumulator.

(e) LDX M - Load register X with contents of memory location M.

(f) STX M - Store register X contents in memory location M.

(g) ADD M - Add contents of memory location M to contents of accumulator and carry if necessary (with carry bit).

(h) AND M - Perform AND operation on contents of accumulator with contents of memory location M and store result in the accumulator.

(i) ORA M - Same as (h), but with OR operation.

(j) EOR M - Same as (h), but with EOR operation.

(k) DEX - Decrement Register X by one.

(l) INX - Increment Register X by one.

(m) CMA M - Compare contents of memory with contents of accumulator. Set the carry flag bit if value in memory is greater than the accumulator value. Set the negative flag bit if value in memory is less than the accumulator value. If the contents of the memory and accumulator are equal, set the zero flag bit. Flag bits are reset for the reverse conditions.

(n) CMX M - Compare contents of memory with contents of register X. Same results as in (m) above.

(o) DEC M - Decrement memory by one.

(p) DEX - Decrement register X by one

(q) PHA - Push accumulator onto stack

(r) PLA - Pull accumulator from stack.

(s) PPS - Push processor status register contents onto stack.

(t) PLS - Pull processor status register contents from stack.

(u) TSX - Transfer SP contents to register X.

(v) TXS - Transfer register X contents to SP.

(2) Control instructions. Instructions in this group are used where decision making operations are required. Included are the following typical instructions.

(a) BRK - Causes the CPU to "break" in its program execution to begin a routine to allow the programmer to enter data or examine the status of the CPU or contents of memory, register or displays. In a typical CPU, the PC register contents are automatically pushed onto the stack. A break instruction is a form of an interrupt under the control of the program instructions. To differentiate between a programmed break and other kinds of interrupts, a flag bit is set which can be examined by a programmed sequence.

(b) JMP P - The PC is reset to the value P, typically the absolute address of the new location where processing is to continue.

(c) BNE N - Branch if the zero flag bit is not set, indicating that the result of some previous operation was not zero in the register involved. The value N, represents an offset (positive or negative) from the program counter. The branch is limited to the maximum allowable values of N but branches beyond this range can be accommodated by using this instruction in conjunction with the JMP P instruction. If no branch occurs, the next sequential instruction is executed.

(d) BPL N - Branch if the result of a previous operation resulted in the resetting of the negative flag bit indicating a positive result. The offset operates the same as in the BNE instruction.

(e) BMI N - Branch by the value of offset N if the result of a previous operation resulted in the setting of the negative flag bit indicating a negative (minus) result.

(f) BVC N - Branch by the value of offset N if the overflow bit is not set (clear).

(g) BVS N - Branch by the value of offset N if the overflow bit is set.

(3) Subroutine instructions. A subroutine is a programming instruction concept that allows a programming sequence written once to be used repetitively any number of times at any point in a program. The subroutine is usually written in a contiguous portion of memory and is accessed by instructions similar to a jump (JMP) instruction, except that the end of the subroutine contains an instruction which causes the program counter to point to the instruction immediately following the one which calls the subroutine. Subroutines may also have within them calls to other subroutines, a process called nesting. When this occurs, the CPU must have a means for remembering which subroutine it is in and to where it must return. Figure IV-3 illustrates the concept of subroutines. In this illustration, the stack and SP are used by the CPU to keep track of the subroutine branch locations. When the instruction to jump to subroutine A is encountered (instruction JSR 0300 starting at address 0204), the CPU places the address of the next instruction the machine must execute after the completion of the subroutine, in this case address 0206, onto the current top of the stack as indicated by the SP. If the first subroutine (A) calls another subroutine (B) as in Figure IV-3, the address of the next machine instruction after the second subroutine call (at address 0310) and placed at the top (next location) of the stack. Since the SP always points at the top of the stack, it always receives the next address it must execute in the subroutine or main program in the proper order. This method of subroutine operation is not the only one possible. Some CPU designs do not employ a stack to hold the return pointers for subroutines. They may use another method which involves placing the return address (next instruction after subroutine return) in a memory location of the subroutine itself, and return to that address by jumping (branching) by the indirect addressing mode. The typical CPU will thus have two basic subroutine instructions:

Address	Instruction or operand		
.			
.			
.			
0100	0206		Portion
0101	0310		of
.	.		Main
.	.		Program
.	.		
0200	ADD		
0201	0500		
0202	STA		
0203	0510		
0204	JSR	Jump to subroutine A	
0205	0300	located at 0300	
0206	LDA		
0207	0230		
0207	0230		
.	.		
.	.		
.	.		
0300	LDA		Beginning of
0301	0331		subroutine A
0302	AND		
0303	0330		
0304	STA		
0305	0330		
0306	BNE		
0307	0020		
0308	JSR	Jump to subroutine B located at 0400	
0309	0400		
0310	RTS		
.		Return to main program	
.			
.			Beginning of
0400	ADD		Subroutine B
0401	0001		
0402	RTS	Return to subroutine A	
.			
.			
.			

Figure IV-3. Subroutines in a program.

(a) JSR M - Jump (branch) to subroutine located at memory address M, and store, either on the stack or in the subroutine, a return address pointing to the instruction immediately following the subroutine call.

(b) RTS - Return from the subroutine to the next instruction after the subroutine. Some types of return instructions may have an indirect address associated with them.

(4) Operational instructions. This group of instructions involves only a single operand whose location is usually one of the registers, including the accummulator or flag bits. The operand stays in its original location after the instruction is executed. Some of the possible instructions of this group include the following:

(a) CLC - Clear (reset) carry flag bit.

(b) CLO - Clear overflow flag bit.

(c) CLI - Clear interrupt flag bit.

(d) LSR Z - Shift contents of register or memory.

(represented by operand Z) to the right by one bit position. The least significant bit is stored as the carry flag bit. This instruction is frequently used to enable the programmer to examine the status of the lowest order bit.

(e) ASL Z - Shift contents of register or memory to the left, storing the most significant bit in the carry flag bit and placing a binary 0 in the least significant position. This instruction is frequently used by the programmer in arithmetic operations such as multiplication routines to shift the multiplicands prior to adding them, as suggested in paragraph II-2.g.

(f) ROL Z - Rotate the contents of a register or memory location to the left by one bit placing the most significant bit in the carry flag and moving the carry flag to the least significant bit location.

(g) ROR Z - Rotate the contents of a register or memory location to the right by one bit placing the least significant bit in the carry flag, and moving the carry flag bit to the most significant location.

(h) COM Z - Provide the complement of a register or memory location. Zeros are substituted for one's and one's for zeros. If the accumulator is the register to be complemented, the carry flag (link) bit is also complemented. This type of instruction is frequently omitted if the CPU has two's complement arithmetic built in. If necessary, this operation can be performed by using a sequence of other instructions.

(i) NOP - No operation is performed. This instruction is useful to the programmer as a "dummy" statement to aid in debugging programs by allowing the later insertion or deletion of instructions or data without upsetting the addressess of other instructions. Another use of this instruction is in time delay routines since the CPU must take one or more clock cycles to execute NOP even though no data is actually altered. Time dependent control processes are applications where this instruction may be employed.

IV-5. Hardware interrupts. An interrupt consists of any action breaking the normal operating sequence of a CPU under programmed control. In paragraph IV-4.b.(2)(a), the Break instruction was discussed. That is an example of a programmed interrupt. A programmed interrupt is used to halt the execution of the CPU and enter a service routine. At other times, it is necessary to interrupt the operation of the CPU to allow it to communicate with the outside world at random intervals through peripheral devices not synchronized with the CPU clock, such as keyboards, printers and A/D converters containing data from control devices. One way to do this is by putting program instructions at regular points in the program causing the CPU to interrogate one by one every device connected to it from the outside world. This is seldom done because it is wasteful in terms of the computer's operating time to interrogate peripheral devices not having anything to communicate to the CPU.

a. Interrupt service routine. The CPU operating speed is often many times that of the fastest I/O devices to which it is connected, making it necessary for the CPU to "wait" for the device to communicate. The hardware interrupt is designed to allow the peripheral device to signal when it is ready to receive or transmit data to the computer. When an interrupt control signal is received by the CPU, it ceases execution of the program and places whatever information is necessary to resume program execution at a later time onto the stack (or elsewhere in memory). The information necessary for the resumption of execution may include the PC, address in the data in other registers, and a return address. The CPU then enters a specified location in memory where the interrupt service routine resides. This routine is a special program containing instructions for transmitting or receiving data from a particular I/O peripheral device. The end of the service routine contains a return instruction to allow the CPU to resume execution of program instructions using the stored information.

b. Polled interrupt. The interrupt process as described thus far is simple to implement on a computer connected to only one peripheral device. However, in the real world, a computer is often connected to a variety of devices, all of which may require interrupt servicing. Since the CPU can perform only one operation at a time, it must determine which device is causing the interrupt signal. It is conceivable that more than one interrupt could occur simultaneously. Also, some devices may have a higher priority than others in terms of system operation. These difficulties may be handled by performing what is known as a polled interrupt. The interrupt signal lines of the various peripheral devices can be wire OR'd together in parallel into one interrupt line. When one or more of the devices request interrupt service,

this line is driven into an interrupt state. The CPU then completes the current instruction, loads the stack with the essential data, and enters the polling routine. At the beginning of this routine, the programmer may or may not set a flag bit to prevent further interrupts from occurring. The polling routine instructions direct the CPU to read the highest priority device's status register through an I/O port address to determine whether it is the interrupting device. If it is not, the next instruction directs the CPU to read the status register of the next highest device, and so on. At times, it may be desirable to allow additional interrupts during the interrupt service routine by nesting, much like subroutine operations. This is allowed when the flag bit which prevents interrupts is reset during the service routine instead of afterwards as it is usually done to permit additional interrupts.

c. Vectored interrupts. The time delay encountered in servicing interrupts is sometimes crucial to peripheral devices,since it may cause the loss of transmitted data. To increase the response time of the CPU in servicing multiperipheral interrupts, the vectored interrupt is sometimes utilized. A vectored interrupt is implemented when the interrupting peripheral device or a wire OR'd common line responds to the interrupt acknowledgement from the CPU by passing the address of the interrupting device's service routine to the PC, thus saving the time required to execute a software polling routine. Of course, more elaborate hardware external to the CPU must be utilized to implement vectored interrupts.

d. Nonmaskable interrupts (NMI). Computers are frequently connected to devices that cannot afford to wait until other interrupts are serviced or to wait during times when the programmer has set the interrupt disable flag. High speed devices such as magnetic disks often fit this restriction. Thus, many computer CPU's are provided with a second interrupt line, the nonmaskable interrupt (NMI). Usually, the highest priority or highest speed peripheral device is connected to this line and its interrupt service requests are answered immediately before any other interrupt, whether or not the interrupt disable flag is set. A vectored interrupt may be used to increase the servicing speed to a NMI request. In many computer systems, software polled interrupts are used for slower peripheral devices such as keyboards and printers, while the NMI line is used for high speed disk storage devices.

IV-6. Arithmetic hardware.

a. Most current CPU instruction sets contain arithmetic instructions limited to addition and subtraction of integers. A computer system must be capable, however, of performing multiplication and division as well as to be able to handle fixed and floating point arithmetic operations. Most computer systems accomplish these tasks through the use of operating system software programs in the form of mathematical subroutine packages or high level languages having such built-in capabilities. A number of routines for dealing with the common mathematical functions such as sine and cosine, exponentiation, absolute values, logarithms, etc, are included in these software packages. These routines (or algorithms) often utilize techniques such as Taylor's expansion or power series to approximate the value of transcendental functions, although solutions to any desired degree of accuracy can be developed.

b. The use of software to perform mathematical operations other than integer addition means that the CPU must execute numerous time consuming instructions when mathematical calculations are required. This, in many instances, severely limits the operating efficiency of CPU in terms of time required to run programs. A possible solution to this is to use a hardware device specially designed to do mathematical operations using high speed logic circuits. CPU speed is increased because the time consuming steps of instruction decoding, PC incrementing and movement of intermediate data between registers or memory are reduced to a minimum. Arithmetic hardware also allows increased pipelining, since the CPU can fetch and decode the next instruction while the arithmetic hardware is performing the complex mathematical manipulations.

IV-7. Input to a computer. To be useful, a computer must be able to receive communications from the outside world through peripheral input/output devices. Since the computer "speaks" in only binary numbers, the peripheral must be capable of translating data meaningful to the programmer or other data source into data meaningful to the computer, and vice versa. Also to be considered is the disparity in operating speeds between the peripheral devices and the CPU. The peripheral device is interfaced to the CPU through an I/O port. Two classes of information must be transferred between the CPU and I/O peripheral devices: data and control (address and status) information. A dedicated I/O bus is generally provided to carry data signals with a separate bus for control signals. The I/O bus may be bidirectional (capable of carrying data in one direction or the other, not simultaneously) or unidirectional (capable of one direction only). Generally, a peripheral device is either an input or an output device, although often both types of devices are packaged in one unit, such as a CRT containing a display for output and keyboard for input. An exception to this is a magnetic disk or tape storage, which functions as both an input and output device. When the highest speed data transfers are required, such as with a magnetic disk, a pair of unidirectional I/O busses may be used employing parallel address and data lines with each bus operating in one direction.

a. Serial I/O interface. Data generally travels between the CPU and the peripheral I/O devices in bit groups consisting of bytes or words. It may also travel in continuous blocks. The fastest way to transmit the data is via parallel lines. However, many peripheral devices operate at low enough speeds to consider the use of serial data transmission methods. Serial transmission requires only a single or pair of I/O data line for half or full duplex operation. The bits are transmitted in a prescribed order and at a fixed rate. In order for the CPU to correctly interpret the beginning and end bits of data from certain devices such as keyboards, start and stop bits are usually placed around the data words. These bits are later stripped off by the CPU during the device interrupt service routines. Data transmission on serial lines is frequently asychronous; it is not synchronized to the CPU clock and communicates at random intervals (but at a constant rate) via interrupts. To provide for interfacing serial transmission between different types of CPU's and peripherals, the EIA RS-232C, a standardized specification for voltage

levels and I/O and control signal pin arrangements, has been adopted. This standard has some serious limitations: its data transmission rate is limited to 20 K bits sec, and it requires two equal voltages of opposite polarity in the 5-25 volt range instead of the single +5 volts of TTL circuitry. To correct these defects, the EIA has introduced two standards, RS-422 and RS-423, that specify TTL voltage levels and a third, RS-449, which is capable of 2 megabits per second. The latter standard is intended to replace the RS-232C. However, the RS-232C is adequate for low speed applications (such as for EMCS) and will probably continue to be used for some time.

b. Parallel I/O interface. High speed devices may be interfaced with parallel interfaces. The IEEE-488 bus is widely accepted as the standard interface device for parallel data transmission. It consists of a 24-line cable carrying 8 parallel data lines and 8 control lines. Applications of the IEEE-488 to EMCS are expected to be unnecessary, since it is used primarily for data rates in excess of 9600 baud.

c. Handshaking. Peripheral devices such as printers and keyboards require a significant amount of time to perform their mechanical functions compared to the rapid operation of the CPU. To avoid delaying the CPU, these peripherals utilize interrupts to signal the CPU when they are ready, as discussed in paragraph IV-5. To indicate when a data transfer operation is complete, either the processor (for input transfers) or the peripheral (for output transfers) signals the other device. This procedure, called handshaking, prevents data transmission before the receiving or transmitting device is ready, which would result in a loss of data. Peripheral devices are usually provided with an input or output buffer to temporarily hold data before it reaches its final destination. The buffer not only isolates the circuit elements of the CPU and peripheral, but holds the data when a busy condition exists. Handshaking signals control the operation of these buffers.

IV-8. Mass storage systems. Although memory storage costs are continually declining, it is still uneconomical to retain more than a single program (or part of it) in the memory directly addressable by the CPU. Since most computer systems utilize a large number of programs and blocks of data, it is necessary to have some means of rapidly transmitting the programs and data to the CPU memory. Several different types of mass storage systems have been developed for this purpose.

a. Magnetic tape systems. One of the earliest forms of mass storage developed, the magnetic tape system is still in wide today. Tape storage is a serial storage medium and therefore not a random access medium like CPU memory. The magnetic tape medium is fabricated from acetate or mylar embedded with minute ferromagnetic metal particles. Data is recorded on tapes using the [illegible] phenomenon, in which a head with electric current flowing through its windings induces a permanent residual magnetic field in metal particles on the tape. The converse electromagnetic phenomenon, in which the moving magnetized particles induce an electric field in the same read head, is used for reading data on

data. The relative motion of the tape and head and the nonlinear magnetization process introduces considerable mechanical and electronic difficulties. Solution of these problems are worthwhile, because of the fact that magnetic tapes are eraseable and rerecordable, making their use highly economical.

(1) Tape recording format. Early tape systems utilized seven read/write heads to produce seven parallel tracks of data across the width of the tape. The trend has been to replace this with nine-track systems. Nine track tape systems allow the recording of a byte or character of data (depending on the coding system used), parity bits or CRC bits to minimize tape recording errors. A few systems use as many as thirteen tracks. Tape data recording densities have continued to increase; a density of 6250 bits per inch (bpi) is in everyday use and densities of 20,000 bpi are under development. However, the most commonly used densities are 800 or 1600 bpi. Between data records, a 3/4" gap is inserted. End of file marks are inserted after a 3" gap at the end of a file.

(2) Encoding methods. A variety of data encoding methods have been employed to record data.

(a) Return to zero (RZ). A "1" is generated by a positive current pulse; a "0" by a negative pulse. The head current passes through zero between pulses. Recording densities are low but ciruitry is simplified.

(b) Nonreturn to zero (NRZ). The current is reversed whenever the bit status changes. However, if a bit is missed, the rest are read incorrectly.

(c) Nonreturn to zero - IBM (NRZI). Current is always reversed on a "1" and never on "0." This method must be used with odd parity to insure the presence of at least one bit for clocking purposes. It is susceptible to skew or tilting of the tape, thereby mixing bits from different characters.

(d) Phase encoding. The most widely used method, it uses the direction of flux transitions to indicate "1's" and "0's." Small flux reversals must be inserted between bit positions to ensure that the proper direction is recorded.

(e) Group coded recording. A modified NRZI method recently introduced by IBM, this method encodes data with extra bit correcting information for lost data.

(3) Advantages and disadvantages of tape. The chief advantage of magnetic tape is high recording capacity. A typical 10.5" (standard reel contains 2400 feet of tape. A 9-track recording at 1600 bpi could contain a maximum of 46 million characters (bytes). However, the actual capacity of this tape would probably be somewhere between 1/2 to 2.5 million characters

due to the presence of numerous 3/4" blank record gaps. Another advantage of tape is the low cost of storage per bit. Although the retrieval of any particular byte of data could take considerable time due to non-random serial storage, the data transfer rate is very high for contiguous blocks of data. Tape is therefore an ideal storage medium for infrequently accessed programs and data, and as a backup storage medium for other mass storage devices.

b. Magnetic cartridge disk. Magnetic cartridge ("hard") disks provide semi-random access operation, because the access time depends on the location of the read/write head relative to the data location on the disk. However, access times are sufficiently rapid for the disk memory storage devices to be considered random access in nature. The memory medium is composed of a number of aluminum disks, typically 14" in diameter coated with iron oxide. The surface of the disk must be extremely flat due to the fact that the gap between the read/write heads and the disk surface may be between 20 - 100 microinches. The rotational speed of a cartridge disk drive is typically between 1500 - 3600 rpm. To maintain a constant data density and transfer rate, only a narrow strip of the disk, approximately 2" wide, is recorded upon. Data is recorded in a series of tracks and sectors. A sector is a portion of a track, and contains header, synchronization, and CRC data for error detection as well as the actual data. A hardware device known as a disk controller is normally included as part of a magnetic disk storage system. Its function is to supervise the recording of data by properly formatting it, preparing header and CRC error data, and maintaining a recorded directory of the contents of the disk. A single disk controller is typically capable of controlling four disk drives. The data capacity of a disk depends upon the recording density, a function of the number of bytes per sector (typically 128-512), the number of sectors per track (typically 12-96), and the number of tracks per surface (typically 32-1450). This results in disk capacities of 2.5 - 44 megabytes per disk. The stacking of disks (disk packs) results in capacities in excess of 300 megabytes per disk system, expandable to multisystem units. Points to consider in selecting disk drives include whether the disks are removable or fixed, top loading or front loading, and average data access times. Magnetic disks have evolved into two basic arrangements:

(1) Moving head disks. In this type of disk drive, the magnetic read/write heads (one per disk surface) move from track to track through the use of rotary positioning coils or highly accurate stepping motors utilizing feedback methods to develop positioning error signals. This type of disk head is currently the most popular.

(2) Fixed head disks. These drives employ multiple nonmovable heads, one per track. The disk and head arrays are commonly supplied in sealed units capable of service in rugged environments where the integrity of data is important. This type of disk drive is gradually becoming more popular.

c. Flexible ("floppy") disk. Flexible disk storage systems were developed to fill the need for a low cost mass storage system to be used with minicomputers having limited storage capacity requirements. The typical floppy disk is a mylar disk 8 inches in diameter and is flexible in comparison with the larger aluminum sub-strated "hard" disks. A single moving head, actuated by a stepper motor travels across 64 to 77 recording tracks of 26 to 32 sectors each. Originally, each sector contained 128 bytes of data; thus a total of 300 K bytes of data could be stored per disk. Presently, the data capacity has been extended with the availability of double sided or double density disk systems (or a combination of both). The floppy disks are always contained within a plastic envelope for protection while mounted on the drive. The read/write head, however, makes contact with the surface recording media of the disk through openings in the envelope, thus limiting the useful life of the disk to considerably less than that of the cartridge disk. A floppy disk controller typically handles up to four disk drives; thus the maximum system capacity is greater than one megabyte with a single disk controller. A floppy disk rotates much slower than a cartridge disk (typically 360 rpm versus 3600 rpm); thus access and data transfer rates are lower. The tracking system for head position is either "soft" (by electronic means) or "hard" sectoring (by means of holes in the disk) depending upon the particular disk system manufacturer.

d. Semiconductor mass storage. The use of semiconductors for mass storage has been inhibited because of the high cost per bit relative to tape and disk storage and the volatility of the memories in the absence of power. Recent technological innovations, however, have indicated that semiconductor mass storage may soon become a reality:

(1) Bubble memory. Bubble memory utilizes the action of magnetic fields on material in an addressable array. Single chips with 500 K bits have been produced and higher capacities are expected. Costs per bit of bubble memory are expected to be competitive with magnetic disk devices. Bubble memory is nonvolatile when power is removed.

(2) Read-only memory (ROM). Read-only memory is a semiconductor device containing software etched-in during the manufacturing process. It is a useful storage medium for system software or often-used programs that are not meant to be altered. The advantage of a ROM is its compactness and the fact that it does not have to be "loaded" onto the computer system; it is immediately addressable as is normal RAM. A ROM is noneraseable; however, a ROM chip can be easily removed from the system and can be replaced by another unit containing other software if desired. ROM's are not produceable by computer users, but are rather produced by the semiconductor manufacturer at the request of a computer or software designer. Individual ROM programming and development costs are high but under high quantity mass production, the cost of a ROM is low. A ROM is not meant to replace other read/write mass storage systems, but can lessen the capacity requirements of such systems to some degree by using them to store important software.

(3) Programmable read-only memory (PROM). In order to enable the computer system builder or user to program his own ROM in the field or in a factory, the PROM was developed. By using appropriate voltages to selectively "blow" resistor fuses, the binary code is permanently imprinted in the PROM device. Great care must be taken in both progamming and encoding the PROM because mistakes are noneraseable. A computer may be used to perform the actual task of entering the desired code into PROM if interfaced to the proper hardware circuitry. PROM's are bipolar TTL devices and are therefore extremely fast. Functions of the PROM are similar to that of ROM.

(4) Erasable programmable read-only memory (EPROM). An EPROM is a PROM with the ability to be erased and rewritten a countless number of times. The read/write times of EPROM's are assymetrical in the sense that it takes much longer to erase the device (typically 15 minutes) than to write on it (1-2 minutes to fill it); thus, an EPROM is not used as a substitute for RAM, only for software storage. The data in an EPROM can be erased only by shining an intense source of UV light through a quartz window on the chip. Individual words or bits cannot be erased, only the data contained on the whole chip. EPROM's have functions similar to ROM's and PROM's.

(5) Electrically alterable read-only memory (EAROM). A desire for an EPROM with more symmetric erase/write times led to the development of the EAROM. This device has the ability to be erased in 10 milliseconds and be rewritten at the rate of 1 millisecond per word. Also, individual words can be rewritten, rather than erasing the whole device. This makes the EAROM useful for program development and debugging, but not as a subsitute for RAM since it is still too slow for that purpose. One defect of the EAROM is that data can be retained for only a finite period of time before it "leaks" away; this retention period can be for as long as ten years, but may decline to only a period of several months if the device is cycled (erased and rewritten) a million or more times.

IV-9. Bus structure. The bus structure of a computer consists of essentially three busses:

a. Address bus. The address bus consists of the parallel lines of number n, where n is the number of bits available to the program counter register in the CPU. The number of memory locations directly addressable by the CPU is related to the number of address lines by the equation:

$$\#\text{ of locations} = 2^n$$

The address bus is attached to every device having a memory address including all of the RAM's, ROM's, etc.

b. Data bus. The data bus consists of n parallel lines, where n is the number of bits of a characteristic word of computer memory. The data bus may or may not be bidirectional. If it is not bidirectional, then separate buses must be used for data traveling to and from the CPU.

c. Control bus. The control bus carries the miscellaneous signals necessary for the proper operation of the computer system. Many of the signals on the control bus are generated by the CPU. Others, however, are generated in other devices, such as peripherals, and are used to notify the CPU of their status. Signals which appear on the control bus include:

(1) Clock pulses. The CPU clock signals are used to control the status of many devices such as memory and peripherals. A positive clock pulse may be used to enable the memory, synchronizing it to the operations of the CPU, ensuring that the logic status of the data will be settled by the time the CPU is ready to read or write it. Clock signals prevent a peripheral from sending data before the CPU is ready.

(2) Read/write (R/W) enable. This control signal prevents any extraneous data from entering or leaving memory. This would happen if data was put on the data bus while the memory's address was on the address bus. The data can enter memory only when the R/W line is enabled. Semiconductor RAM chips commonly have a R/W pin for this control signal.

(3) Memory bank select. These lines are used to select a particular memory bank. Normally, the computer memory is subdivided into banks, each having less than the maximum number of address lines. The excess lines enter a decoding circuit to convert the binary address into a memory bank enable signal. Some computer memory banks may have their own address decoding circuitry, making these control lines unnecessary.

(4) Interrupt lines. The maskable interrupt and NMI signals, as previously discussed, are connected to the CPU via these lines.

(5) Reset line. A signal from an external device may provide a reset signal to initialize the CPU and its internal registers.

(6) Bus available. The CPU provides this signal to indicate to an external device, such as a direct memory access (DMA) controller (to be discussed in Section VIII), that the CPU is stopped or performing internal operations and the data and address busses may be used for purposes other than CPU program execution.

(7) Ready. If the CPU is connected to the memory with slower access times such as in PROM's and EAROM's, the CPU may be delayed for a single clock cycle by enabling the ready line. This allows time for the data from the slower memories to stabilize.

Section V. ASSEMBLY LANGUAGE PROGRAMMING

V-1. Machine language. A digital computer operates by utilizing binary numbers alone. All of the operations included in its instruction sets are recognizable to the computer only as a binary code. The instructions such as Add, AND, Jump, etc., must be translated to their binary code to be interpreted by the machine. Since data is of the same binary form as the instruction codes, extreme care must be taken to prevent the CPU from interpreting numerical data or addresses as instruction code and vice versa. There are times, however, when it is desirable for the CPU to do this when instructions are themselves being modified by the machine. To illustrate some basic machine language programming concepts, an abbreviated version of the instruction set of an 8-bit microprocessor CPU, the 6502 developed by MOS, Inc., is shown in Table V-1. Only a few instructions and addressing modes are included for simplicity. In the example program shown in Table V-2, two numbers from memory locations 100_{10} and 101_{10} are added together and the result is stored in location 102_{10}. In this simple program, the CPU would begin execution at locations 0000 and proceed through location 0009_{10} where it would halt awaiting external instructions. The CPU reads the first instruction at 0000, 10101101, which it interprets as an instruction to load the accumulator with the contents of the memory at the location given by the two bytes immediately following at 0001_{10} and 0002_{10}. It is a peculiarity of the 6502 CPU that the least significant address byte is required before the high order address byte (the 6502 has a 2-byte or 16-bit address bus). Thus, the two address bytes taken in reverse order read ("00000 00001100100"), equalling 100_{10}, the address of the first memory location of interest. The contents of this memory address, 5, is loaded in the accumulator. At this point, the program counter is at 0003, where the next instruction is located. The CPU interprets this instruction as "Add the contents of the address given by the next two bytes (absolute address) to the accumulator." The next two bytes give the address 101_{10}, containing 14_{10}. This value is added to the accumulator, which now contains 19_{10}. The next instruction at 0006 directs the CPU to store the contents of the accumulator at 102_{10}. The accumulator and memory location 0102_{10} will then both contain 19_{10}. The last instruction in the program, 00000000, directs the CPU to halt. Notice that this instruction is numerically the same as bytes located at addresses 0002, 0005 and 0008, but the CPU interpreted these locations as addresses rather than instructions. This example, although extremely simple in function, is tedious to write out because of the use of binary numbers to express machine operations. Several methods can be employed for simplifying this process.

Table V-1. Simple computer instruction codes for an 8-bit CPU.

Mnemonic	Instruction	Address Mode	Binary Code
ADC	Add to A	Immediate	01101001 xxxxxxxx
ADD M	Add M to A	Absolute	01101101 xxxxxxxx xxxxxxxx
LDA	Load A	Immediate	10101001 xxxxxxxx
LDA M	Load M to A	Absolute	10101101 xxxxxxxx xxxxxxxx
STA	Store A in M	Absolute	10001101 xxxxxxxx xxxxxxxx
BRK	Break (halt)	Implied	00000000

SYMBOLS

A - Accumulator
M - Memory location
x - binary 0 or 1, depending on memory contents

Table V-2. Example machine language program for an 8-bit CPU.

Memory Location*	Binary Contents	Comments
0000	10101001	Load A (Absolute)
0001	01100100	Low Order Address
0002	00000000	High Order Address
0003	01101101	Add M to A (Absolute)
0004	01100101	Low Order Address
0005	00000000	High Order Address
0006	10001101	Store A at M (Absolute)
0007	01100110	Low Order Address
0008	00000000	High Order Address
0009	00000000	Halt
.		
.		
.		
0100	00000101	Contents of 0100 = 5_{10}
0101	00001110	Contents of 0101 = 14_{10}
0102	00010011	Contents of 0102 = 19_{10}

*Memory address location in base 10

V-2. Hexadecimal coding.

a. From Section II it was shown that the octal and hexadecimal number systems could be used to simplify binary representation. Either of these methods could be used to simplify the programming example of the previous paragraph. The hexadecimal number system is employed here for the reason that it was chosen by the 6502 microprocessor manufacturer because it is more compact than octal and better fits the byte oriented architecture of the 8-bit CPU. Table V-3 shows the same program of Table V-2 translated into hexadecimal representation. The program is now less tedious to write but the awkwardness of binary numbers has been traded for the difficulties associated with the hexadecimal representation.

b. Programmers usually do not write their programs in the column format shown thus far because it is easier to write the instructions and operands together. This helps the programmer avoid leaving out part of an address when he is dealing with absolute addresses versus relative or indirect addressing requiring fewer bytes. Some instructions, BRK for example, are one byte instructions for an 8 bit CPU. Table V-4 shows the example machine language program of Table V-3 written with instructions and operands grouped together.

Table V-3. MACHINE LANGUAGE PROGRAM IN HEXADECIMAL

Memory Location (Hex)	Hexadecimal Contents	Comments
0000	AD	Load A (absolute)
0001	04	Low Order Address
0002	00	High Order Address
0003	6D	Add M to A (absolute)
0004	65	Low Order Address
0005	00	High Order Address
0006	8D	Store A at M (absolute)
0007	66	Low Order Address
0008	00	High Order Address
0009	00	Halt
.		
.		
.		
0064	05	Contents of 10010 = 5_{10}
0065	0E	Contents of 10110 = 14_{10}
0066	13	Contents of 10210 = 19_{10}

Table V-4. Machine language program with grouped instructions.

Memory Location (Hex)	Hexadecimal Instruction and Operands
0000	AD 64 00
0003	6D 65 00
0006	8D 66 00
0009	00
.	
.	
.	
0064	05
0065	OE
0066	13

V-3. Assembly coding.

a. The machine language coding procedures shown thus far are still tedious in view of the fact that only a few instruction codes have been considered for use in a short, simple program. The full instruction set of the 6502 microprocessor, for instance, contains 54 instructions. With all addressing modes considered, a total of 151 instructions and their operating codes may be used. The programmer must either memorize every operating code and remember its function, or else look up the proper code for every instruction as programs are written. Added to this task is the requirement that the programmer must assign and keep track of every address where data is to be stored. Should the programmer be forced to change some of the program coding, as often happens, some or all of the data memory locations may have to be changed. In a long, complicated program, this can be a time-consuming and confusing task. To alleviate these problems, a program called the assembler has been written to enable the programmer to use the computer itself to do many of the tedious bookkeeping tasks associated with machine language programming.

b. To utilize the assembler, the programmer must write the program in a special format known as assembly language. Instead of having to remember the actual hexadecimal OP codes, the programmer uses simpler 3-letter (upper case) mnemonics to indicate machine instructions followed by lower case letters or character symbols to differentiate between addressing modes where necessary. Tables V-5 and V-6 summarize the operating instruction mnemonics and addressing mode characters for the 6502 assembler. It is noted that each type of CPU will have its own set of mnemonics and symbols for its own assembler program. As an example, to indicate an instruction for the 6502 such as "load immediate to accumulator the value FF_{16}," the programmer writes:

```
LDA# FF
```

Some combinations of addressing modes are allowed in the 6502 microprocessor. For example,the instruction "Store accumulator in memory indirectly indexed with the base operand EA_{16} and index register Y" is given in assembler format:

```
STAiy  $EA
```

The interested reader should consult the manufacturers literature or various texts detailing the various addressing modes available for the 6502 instructions. Assemblers for other microprocessors differ in details but not in the basic format described.

TABLE V-5. INSTRUCTION MNEMONICS

Mnemonic	
ADC	Add memory to A with carry
AND	"AND" memory M with A
ASL	Shift left 1 bit (M or A)
BCC	Branch on carry flag clear
BCS	Branch on carry flag set
BEQ	Branch on result zero
BIT	Test bits 6&7 in M with A
BMI	Branch on result minus
BNE	Branch on result not zero
BPL	Branch on result plus
BRK	Programmed interrupt
BVC	Branch on overflow flag clear
BVS	Branch on overflow flag set
CLC	Clear carry flag
CLD	Clear decimal mode
CLI	Clear interrupt flag
CLV	Clear overflow flag
CMP	Compare M and A
CPX	Compare M and index X
CPY	Compare M and index Y
DEC	Decrement M by 1
DEX	Decrement index X by 1
DEY	Drecrement index Y by 1
EOR	Exclusive "OR" M with A
INC	Increment M by 1
INX	Increment index by 1
INY	Increment index Y by 1
JMP	Jump (unconditional)
JSR	Jump to subroutine
LDA	Load accumulator
LDX	Load index X
LDY	Load index Y
LSR	Shift right 1 bit (M or A)
NOP	No operation
ORA	"OR" M with A

Mnemonic	
PHA	Push A on stack
PHP	Push CPU status on stack
PLA	Pull A from stack
PLP	Pull CPU status from stack
ROL	Rotate left 1 bit (M or A)
ROR	Rotate 1 bit right (M or A)
RTI	Return from interrupt
RTS	Return from subroutine
SBC	Subtract M from A w/borrow
SEC	Set carry flag
SED	Set decimal mode
SEI	Set interrupt disable flag
STA	Store A in M
STX	Store index X in M
STY	Store index Y in M
TAX	Transfer A to X
TAY	Transfer A to Y
TYA	Transfer index Y to A
TSX	Transfer stack pointer to X
TXA	Transfer X to A
TXS	Transfer X to SP

Abbreviations

A - Accumulator
M - Memory location
SP - Stack pointer
X - Index X
Y - Index Y

TABLE V-6. Addressing mode codes for 6502 MPU assembler.

Symbol	Addressing Mode
@ or a	Absolute
#	Immediate
$	Hexadecimal number
i	Indirect
r	Relative
y	Y index
z	Zero page

c. In the example of assembler statements shown above, the operands were shown as hexadecimal codes. They are necessary when the programmer wants to indicate constants or initial values of data in the program. When, however, the operand is intended to designate a memory location, it is not necessary to indicate an actual address, but merely a unique alpha-numeric symbolic "name" to stand for the location of a defined variable. The maximum number of alphanumeric characters the programmer may use for a symbolic memory location name is a function of the particular assembler being used. Usually, six or more characters are permitted, allowing more than enough unique variable names for even the largest program. Additional operations meaningful to the assembler may be added on at the end of a symbolic address. For example, the instruction shown below tells the CPU to store the contents of the accumulator at the zero page address calculated from the memory address assigned by the assembler, to POINTH plus 1, added to the value of the index X:

```
STAzx    POINTH+1
```

d. In the previous examples shown, the programmer was required to keep track of the memory address locations of the program itself. This was necessary to prevent the programmer from placing data where program code exists and vice versa. The assembler can eliminate this chore, thereby requiring the programmer to provide only symbolic labels for special addresses, such as the beginnings of subroutines or branch points in the program. The assembler will then assign addresses in the program subject to the memory space available and instructions from the programmer indicating where in memory the program is to begin. It is a desirable property of a program to be able to move it to another point in memory from where it was originally written. Programs with this property are said to be relocatable, and they enable the programmer to move them about in memory without worrying whether or not they will execute properly. Programs of this type make extensive use of relative addressing. Some assemblers will produce code of this type, but the programmer must be sure to use instruction modes that do

not require absolute addressing. The assembler also allows the programmer to write non-executable comments alongside the assembly language statements by preceding them with a character, usually a semicolon, which delimits or separates the comments and instructions. Thus, the typical assembler requires a program statement format as shown in Table V-7. A program made up of such statements is known as source code.

TABLE V-7. Typical assembly language statement format.

Label	Instr.	Operand	;Comment
			Optional-used to make the program logic more under standable.
		May or may not be needed by instruction. The Operand can be one or two Hex bytes or asymbolic name indicating an address*	
	Instruction-A 3-letter mnemonic sometimes followed by other symbols differentiating between addressing modes.		
An optional name in alphanumeric code to indicate a particular reference memory location. May be omitted if location is not significant in program.			

*This statement applies to an assembler for a CPU with an 8-bit word. Other word length machines may have different operand parameters.

e. It is the purpose of the assembler to take the source code submitted by the programmer and transform it into machine executable object code. Assemblers differ in their ability to perform this task. A one-pass assembler can take the assembly language program into executable code in a single step. To do this in one step requires a considerable amount of memory space in the computer because the assembler must scan the complete program to allocate and assign memory locations and prepare symbolic names designated by the programmer and addresses assigned by the assembler. A one-pass assembler produces an output consisting of the object code side by side with the corresponding assembly language statement (including comments). If any program statement coding errors are present, most assemblers will also produce a copy of the symbol table for the convenience of the programmer. Assemblers that accomplish the same functions with two or more passes require less memory space but are slower and more tedious to use. They require the programmer to feed the output of the first pass back through the computer along with other portions of the assembler and the original program. The object code, when produced, may thereafter be used to execute the desired program without further action by the assembler. However, if the programmer desires to change any of the program statements, the program must be re-assembled.

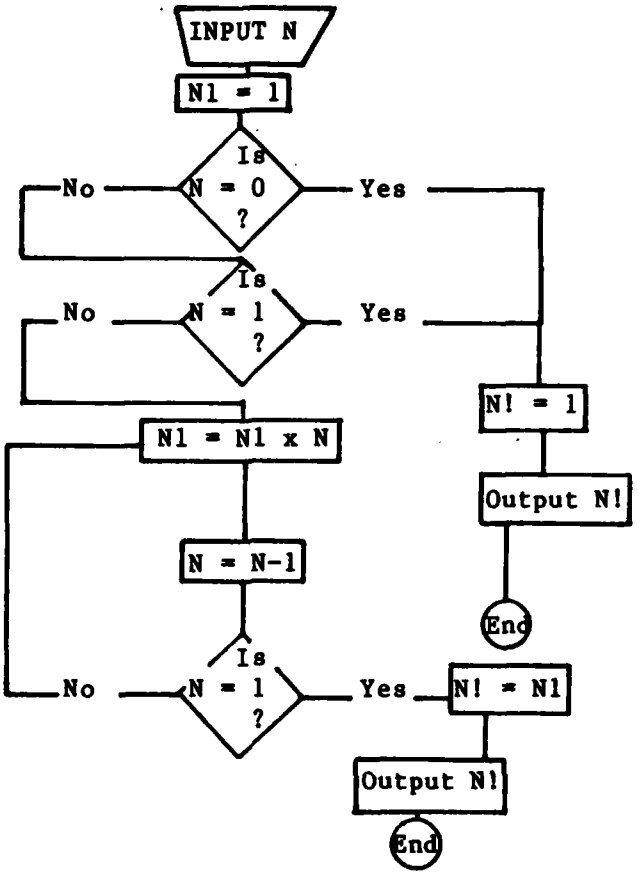


Figure V-1. Flow chart for program to calculate N factorial.

V-4. Programming procedures. It is difficult to teach programming procedures without actually practicing the programming. However, several techniques are presented concerning the philosophy of programming.

a. Flow charting. A useful technique available to the programmer is flow charting. This procedure consists of preparing a logical diagram of the sequence of operations the CPA must perform to solve the problem at hand. Programmers commonly enclose logical or arithmetic statements in a rectangular box, and use diamond shaped boxes to enclose statements in which a choice is to be made in the program by branching. A typical flow chart is illustrated in Figure V-1 showing the logic for a program to calculate the value of the familiar factorial function for integer N. Note that the logic directs the program to first check to see if the value of N is 0 or 1. In that case, the value of N is defined as 1 and the computer immediately outputs that result. For other values of N, the computer multiplies decremented values of N successively until N equals 1. Note the use of the intermediate variable, N1. The flow chart logic solution to any problem, especially the more complicated ones, are by no means the only ones possible.

(1) Each programmer develops a unique aproach to problem solving. However, some solutions are better suited to a particular CPU and a programmer may take advantage of CPU features to develop a solution capable of being executed faster or more accurately than others.

(2) The programmer must also take into account the limitations of the CPU with regard to the maximum size of numbers, such as integers, that the computer can represent. For example, an 8 bit microprocessor CPU can represent integers no larger than 255 unless more complicated software techniques are used.

(3) The example flow chart of Figure V-1 is not detailed enough for the assembly language programmer, since the program must also contain routines instructing the CPU how to multiply. Routines such as that can be indicated as subroutines on flow charts. High level languages, to be discussed in the next section, may utilize flow charts such as that of Figure V-1.

b. Algorithms. Many standard procedures for obtaining common mathematical quantities have been developed in routines known as algorithms. The programmer may either obtain algorithms written specifically for the particular CPU being used, translate them from the algorithms of other machines or write them from mathematical procedures. Such algorithm packages may include routines for multiplication and division, trigonometric functions, fixed and floating point numbers. Each CPU can perform single precision arithmetic by using a single or double byte to represent the value of numbers. When more accuracy (greater number of digits) is needed, double and triple precision algorithms exist, to be incorporated by the programmer into programs.

c. Text editing. The actual process of writing programs into memory can be a difficult and time consuming task on the assembly language level. Text editor programs allow the programmer to write and manipulate the program statements more rapidly. Statements may be added, deleted or changed by simple commands and the editor can be made to keep track of addresses and symbolic names. Text editors are frequently associated with assemblers.

d. Documentation. Information consisting of program source and object code, comments, flow charts, useful data, operating procedures and applications, makes up what is known as documentation. This information is important not only to the original programmer, but especially to anyone else who may utilize the program. Since programs are rarely "perfect" in the sense that no errors exist for untried conditions, a programmer must occasionally delve into the code to "debug" the program. Without proper documentation, this task can become impossible.

e. Disassemblers. When no documentation exists for an object code program, it is possible to create an easier way to understand assembly code representation of the program (without the original comments) through the use of a disassembler. The disassembler is a program that reverses the process of an assembler by outputting OP code mnemonics and operands from the raw machine code.

f. Cross-assemblers. It is sometimes desirable to develop programs on one particular computer for use on another containing a CPU with completely different architecture. This can be accomplished by means of a program called a cross-assembler. A cross-assembler is dedicated to transforming assembly language code written for a specific host computer into object code executable on one particular CPU. Thus, a mainframe computer could potentially be used to create and debug programs executable on a microprocessor system. This eliminates the need for separate assembler software for the microprocessor, and the detailed knowledge for programming it. Standardized algorithms can thus be rapidly made available to a wide range of CPU's through the use of cross-assemblers.

g. Microprogramming. Microprogramming is a combined hardware/software procedure enabling the programmer to, in effect, create an instruction set based on simpler or "primitive" CPU instructions. Microprogramming is usually implemented by storing in ROM the newly defined instruction codes and their corresponding algorithms. This ROM, known as the control store, contains sequences of simple machine operating instruction codes utilizing only a few different instructions to make up the complete set of CPU instructions. Thus, for every single micro-instruction utilized by the programmer, the CPU internally undergoes several cycles of micro-instructions. The programmer is normally oblivious to the internal CPU operations directed by the control store. Some more recent CPU's enable the programmer to microprogram the CPU and make changes in the microcode at will.

Section VI. HIGHER LEVEL LANGUAGES.

VI-1. Purpose.

a. The concept of assembly language programming was developed in an effort to make the programmer's job less tedious by utilizing the power of the computer itself to convert simple instruction mnemonics and symbolic variable and address names into machine code. Assembly code is very efficient in terms of the compactness of program memory storage requirements and executing speed, providing that it is well written. However, complex programming problems still take an inordinate amount of time and effort by specialized programmers skilled in the use of assembly lanaguage. It is for that reason that the concept of higher level languages was developed.

b. High level languages are problem oriented; no single language is ideal for all applications. Some languages are written to facilitate the solution of scientific and engineering related mathematical problems, while others are more adept at performing business oriented functions such as filing and sorting of alphanumeric data or bookkeeping.

VI-2. Language structure.

a. Statements. Each high level language is provided with a limited set of statements, each defining a single operation. In general, these statements are English-like words or phrases, to aid the programmer in recognizing them and remembering their function. The statements may have numbers associated with them to act as statement ordering numbers (or sequence numbers) or labels, similar to assembly language statements. Other numbers may appear as operands. The statements themselves are generally more powerful than individual machine instructions; a single high level language statement might require a lengthy sequence of machine instructions to be executed.

b. Types of statements. To be useful, high level languages must have some or all of the following types of statements:

(1) I/O statements. The programming language must provide a means of getting data into and out of the computer. It must be capable of allowing more than one hardware data source to communicate with the machine, such as a keyboard and a magnetic disk. The format of the data must be recognizable and controllable in the high level language. For example, the programmer must be able to utilize integers, floating point and fixed point numbers, and designate in the program the format and location in which the output quantities are to be placed on the output medium. The high level language internally decodes the form by which the computer stores numbers and converts them to the format required by the programmer. Thus, the high level language must be tailored to the specific architecture of the machine upon which it is implemented.

(2) Arithmetic and logical operations. Most languages allow the programmer to perform arithmetic operations such as add, subtract, multiply and divide, and compute higher level functions like exponentiation, roots, trigonometric functions, modulus arithmetic and absolute values. Constants may be stored in the program itself. Instead of single operations, the programmer may utilize single complex algebraic statements. The programmer specifies variables by the use of symbolic names rather than memory addresses. The high level language contains internally the complicated algorithms for processing the mathematical functions to arrive at solutions to a high accuracy within the limitations of the machine architecture. Logic operations such as "equals," "greater than," "less than," "AND" and "OR" are frequently included in high level languages.

(3) Subroutine statements. An important ability for high level languages is to allow the programmer to use subroutines. Subroutines are useful to the programmer when procedures which are analogous to each other are used at several points in the program. The method of subroutine operation allows the passing of different data to variables each time a subroutine is called. The procedure of jumping to a subroutine and returning to the next executable statement is the same in high level languages as in machine and assembly code execution.

(4) File or memory declaration statements. The high level language program must indicate to the computer how much memory it will require on line or in the mass storage devices. Every computer system is limited by the number of memory or storage locations it can provide. Programmers sometimes have to reorganize the data handling portions of their programs when these limits are exceeded.

c. High level languages versus assembly language. High level languages are easy to use, but are generally less flexible than assembly language. In addition, much more elaborate computers with larger memories must be used with high level languages because they require the use of extensive software packages generally residing on magnetic disks. Assembly language software, on the other hand, may reside entirely on active RAM and be loaded into memory from magnetic or paper tape. Some assemblers reside permanently on line in ROM's. Computer systems capable of high level language operation are thus more costly than assembler oriented systems. However, the increased costs can often be traded-off due to the ease of use for less highly trained programming personnel and the inherently higher productivity afforded thereof. There are some instances which preclude the use of high level languages, thus requiring assembly coding. These include situations in which memory available to the CPU is highly limited; situations in which CPU execution timing is critical to real-time events; and situations in which software must provide critical protocol or handshaking signals to devices not provided for by high level languages.

VI-3. Compilers. Like assembly language, high level languages require translation from the English-like statements recognizable to the programmer into machine code executable by the CPU. The compiler is a program designed to perform this function. Like the assembler, the compiler may be one-pass or multi-pass in operation. Most recent computer systems use one-pass compilers. The compiler is a large, highly sophisticated program by comparison with the assembler. It performs several functions while operating on the high level language program:

a. Syntax analysis. Syntax refers to the precise format requirements for statements in a program written in a high level language. Those requirements include proper spelling of instructions in the set recognized by the language compiler, proper location of operands and symbolic labels, correct representation of numbers, definition of variables, and other rules peculiar to the language used. The program is first checked for syntax by the compiler. Improper statements cause the compiler to issue error messages corresponding to the statements and cease execution. Some compilers do not find all program errors the first time, since improper syntax may conceal errors in other statements. Complete elimination of syntax errors may thus become a time consuming task requiring several tries. The compiler will not begin to generate machine code until it is satisfied as to the correctness of the program syntax.

b. Intermediate code. Many language compilers generate an intermediate assembly code before producing the final executable code. The compiler must go through the same steps as the assembler in developing a symbol table. In addition, the compiler must allocate space in the computer memory in accordance with statements in the program. At times, the program or its memory requirements will exceed the RAM locations addressable to the CPU. In this case, some compilers have the ability to segment the programs and operate via overlays. Segmentation involves breaking up the disk stored program into smaller pieces without upsetting the logic flow. This allows the CPU to overlay, or bring various segments in and out of memory from disk as required by program logic. This process slows down the execution of the program considerably since large program and data blocks are involved, but it enables the computer to handle programs of virtually unlimited size. Computer memory may also be extended by the concept of virtual memory. In this scheme, programs or data in mass storage are given an address partially corresponding to the smaller active RAM. A hardware controller allows the programmer to access data in mass storage as if it were in RAM.

c. Linkers and loaders. Main portions of programs and their subroutines are generally compiled separately. Programs referencing built-in functions such as sine and cosine or square roots cause the compiler to add the required function algorithms to the program assembly code. When all of the program is compiled into assembly code, all of the various components, the main program, subroutines and functions must be connected together into a logicallly consistent package. This operation is performed by the linker program in the compiler. When this is complete, the compiler translates the assembly code

into object code. Some compilers eliminate the assembly code and translate directly from high level language to object code. Most compilers attach a routine to the object code called a loader enabling the CPU to load the object code from mass storage into RAM before execution. As a final step, the compiler writes the object code into mass storage to await execution. Programs too large to fit in the available memory are frequently compiled by a type of segmentation process. If not, an error will result and the program will not be compiled. The object code generally takes up far less memory space than the high level source code.

d. Execution. At the successful completion of the preceding step, the program is ready for execution. There is no guarantee, however, that it will execute properly as a result of possible logic errors in programming. For example, a common error is one in which the program attempts to make the computer divide a number by zero. This can happen when the programmer does not allow for variables used in the denominators of algebraic expressions occasionally becoming equal to zero, a situation avoidable by careful programming. In a division by zero, the computer operating system monitor will cause the CPU to enter an error routine and halt execution, "bombing" the program. The first execution of a program is the beginning of the debugging process, possibly requiring the rewriting and recompilation of all or part of the program by a process called link editing.

VI-4. Interpreters. An alternative exists to the compilation of high level languages as described in paragraph VI-3, above. An interpreter is a software program that translates a high level language program line by line into executable code. This process is much less efficient in terms of execution time than by the compiler method and requires far more memory, since the source code is not transformed into stored object code. However, when source codes are being debugged or being used in an interactive environment, interpreters are very effective. Interactive environments are used when a programmer wishes to execute programs in a conversational mode with the computer. Keyboard/printers and CRTs are commonly utilized in such cases in time-shared environment. Interpreters have an additional advantage in that the software is usually much smaller than that of a compiler for the same language. Frequently, interpreters are placed as permanent on-line software resident in ROMs.

VI-5. FORTRAN.

a. General. The most popular scientific and engineering high level language in use at present is FORTRAN, a mnemonic name standing for FORmula TRANslation. This language is equipped with a set of statements enabling the programmer to solve most kinds of problems. Some of these statements are presented. Many more have been added, but these included are the basis of the language.

b. Statement format. FORTRAN statements vary in their format, but must be written conforming to common column location requirements. The statements are written up to 80 characters wide. The first five columns (characters) are allocated for any optional numerical labels. The sixth column is reserved to indicate a continuation; statements too long for the 65 available character columns may be indicated on the next program line. The program statement area extends from column 7 to column 72. The last eight columns may contain sequence numbers, used only in line editing by batch systems.

c. I/O statements. Input and output devices are indicated by assigning hardware to integer logical unit numbers. The basic I/O statements include:

(1) READ. The READ statement has a format as shown below: Label

```
READ (m,n) VAR1, VAR2,...VARN
```

The "n" in the above statement is the logical unit number of a hardware device. By convention, a card reader or keyboard is given a logical unit number of 5. The operands VAR1 through VARN represent "N" variable symbolic names assigned to memory locations by the compiler/interpreter. The label refers to an optional number assigned by the programmer to allow branches from elsewhere in the program to that point. The symbol "m" refers to the FORMAT statement label.

(2) WRITE. The WRITE statement has a format identical to the READ statement and allows output to various logical devices. By convention, the printer is assigned a logical unit number of 6.

(3) FORMAT. READ and WRITE statements are accompanied by FORMAT statements as shown below:

```
Label    FORMAT (a,b,...)
```

The symbols a,b,... refer to parameters directing the compiler or interpreter to properly space alphanumeric data and to represent numbers as integers, fixed or floating point quantities, determining where the decimal point, if any, is to be. A "PRINT" statement is sometimes also used in place of WRITE and FORMAT statements in some newer versions of FORTRAN.

(4) File declaration statements. The format of these control statements varies from computer to computer. They assign a particular data file to a logical unit number which may appear in READ and WRITE statements.

d. Decision making statements. A number of decision making statements are available in FORTRAN:

(1) IF. The IF statement has two formats as shown below:

```
Label   IF (arithmetic and/or logic statement) k,m,n,
Label   IF (arithmetic and/or logic statements) algebraic
        equation or GO TO n
```

The arithmetic or logical statements inside the parentheses may use two or more symbolic variables separated by arithmetic or logical symbols including +, -, * (multiply), / (divide), .EQ. (equals), .LE. (less than or equal), .LT. (less than), .GE. (greater than or equal to), .GT. (greater than), .NE. (not equal to), .AND., and .OR. functions. The symbols k, m, and n refer to statement label numbers to branch to if the result in the expression in the parenthesis is negative, zero or positive, respectively.

(2) GO TO. The GO TO statement has two different formats. The first is an absolute GO TO corresponding to an absolute addressed jump, and the second is a computed GO TO, corresponding to an indirect jump. The format is as shown below:

```
Label  GO TO N
Label  GO TO (LABEL1,...,LABELN),N
```

In the second statement, the value of variable integer "N" existing at the time the statement is executed determines the statement number to which the branch occurs.

(3) DO. While not exactly a decision making statement, the DO statement allows repetitive looping until a counter is satisfied. The format of the DO statement is as shown below:

```
DO  LABEL    J - M,N

Label CONTINUE (or most other statements)
```

The statement indicated by the LABEL is the last statement in the DO loop. The repetition counter is J, with an initial value of M and a final value N. Both M and N may be constants or variables. CONTINUE is the end statement of a DO loop; however, more recent versions of FORTRAN allow DO loops to end on most other labeled statements.

(4) STOP and END. Program break statements. The STOP causes a halt and an END terminates execution. These statements are required at the end of the main program. The end of a subroutine requires an END statement.

e. Subroutine statements. A subroutine may be called by using the following statements:

(1) CALL. The CALL statement has the format shown below:

```
CALL SUBROUTINE LABELNAME (A,B,...)
```

The LABELNAME is a unique symbolic name for the subroutine. The symbols A,B,... refer to various values of variables passed to and from the subroutine. Other subroutines may be called while in the subroutine.

(2) RETURN. The subroutine is terminated by the RETURN statement (followed by END). This statement has no operands. Its effect is to cause a transfer back to the program statement immediately following the subroutine CALL statement.

f. Memory and other declaration statements. Several statements are used to allocate memory and declare the variable types:

(1) DIMENSION. The DIMENSION statement is used to allocate memory to multidimensional arrays. The format is as shown below:

```
DIMENSION VARNAME1(N,...), VARNAME2,...
```

The symbol(s) N refers to the number of variables in row, column, etc. of the array.

(2) REAL. The REAL statement declares variables with symbolic names beginning with either I, J, K, L, M, or N to be real numbers. Normally, only integer variables begin with one of these six letters (all symbolic names must begin with a letter). The format of this statement is as shown:

```
REAL INAME, JNAME,...
```

(3) INTEGER. The INTEGER statement is similar to the REAL statement. It declares that the variables following it beginning with the non-integer characters (any letters but I, J, K, L, M. and N) are to be considered integers.

(4) DATA. This statement allows the programmer to insert numerical constants into the program assigned to variable names as single values or indexed multidimensional arrays.

g. Arithmetic statements. FORTRAN allows algebraic expressions to be evaluated. This is done by setting an equivalence between a variable label name on the left side of an equals sign, and an algebraic expression on the right side made up of constants, arithmetic symbols, and variable label names. The variable quantity on the left is set equal to the quantity on the right through the use of an equals sign; thus, its value set equal to that of the evaluated algebraic expression.

h. Efficiency of the FORTRAN compiler. The efficiency of FORTRAN object code in terms of compactness and speed is less than that of assembly code written by a skilled programmer. The differences are largely machine dependent; however, compilers of different machines produce varying results. Although FORTRAN is a universal, standardized language, programs can rarely be run on different machines without some modification. All high level languages are, therefore, hardware dependent.

VI-6. BASIC.

a. General. A second very popular high level language for scientific and engineering applications is BASIC. In many ways, BASIC is similar to FORTRAN, although it is not as powerful. The chief difference between BASIC and FORTRAN lies in the fact that BASIC is almost entirely an interpreted language while FORTRAN IS predominantly a compiled language. The BASIC language is, therefore, more easily implemented on smaller computer systems, frequently resident in the system on ROM. Deficiencies of computers using interpreted BASIC include their slower execution time and large memory space requirements. BASIC is also deficient in its ability to handle data files, on magnetic disk, although more recent enhanced versions of the language have added this capability.

b. BASIC statements. The commands in the BASIC repertoire are briefly summarized:

(1) DIM. Corresponds to the FORTRAN dimension statement.

(2) IF...THEN. Corresponds to the FORTRAN IF statement.

(3) LET. This statement corresponds to the FORTRAN algebraic equivalence statements.

(4) FOR...TO. Corresponds to the FORTRAN DO statement.

(5) NEXT. Corresponds to the CONTINUE or other end statement of a FORTRAN DO loop.

(6) GOSUB. Corresponds to a FORTRAN subroutine call. The subroutine is designated by the program line statement number (all BASIC statements are numbered).

(7) RETURN. Same as RETURN in FORTRAN.

(8) INPUT. Corresponds to a FORTRAN READ statement.

(9) READ. Used to read BASIC DATA statements.

(10) DATA. Contains data constant values.

(11) PRINT. Corresponds to FORTRAN WRITE and PRINT statements. Basic does not use FORMAT statements but allows some latitude to the written output by using tabs for placement and built in scientific notation.

(12) END. Corresponds to FORTRAN END statement.

(13) REM. Corresponds to a CONTINUE statement in that it is a dummy statement, frequently used to enter comments into the program for documentation purposes.

VI-7. Other languages. As many as 200 distinct computer languages have been developed over the years by various manufacturers, universities, and users. Of these, a small number have achieved widespread popularity:

a. COBOL. COBOL was developed primarily as a business oriented language. It is especially useful for manipulating files containing various types of data. COBOL is implemented on most mainframes and minicomputer systems.

b. ALGOL. This language was the first of the so-called structural languages. ALGOL combines aspects of COBOL and FORTRAN, thus classifying it as a powerful all purpose language. Originally developed by the Burroughs Corporation for its own line of computers, ALGOL has been implemented on many other mainframe and minicompuer systems. It sufers from the deficiency that I/O statements are not defined and must be individually developed for each new type computer system.

c. RPG. Introduced originally by IBM, this language is wholly oriented for business use. It is especially useful in generating and formatting reports.

d. PL/I. This language represents a family of similar FORTRAN-like languages developed for use with IBM Corporation computer systems.

e. APL. Unlike most other languages, this language was developed to be used in an interactive environment. It is suited towards the solution of mathematic problems in engineering and scientific work. Very powerful mathematical tools are built into APL. For example, a matrix can be inverted by invoking a single command.

f. PASCAL. The most recent development in structured programming, this language bears a strong resemblance to ALGOL. Like ALGOL, it suffers from a lack of definition in its I/O statements. The language features powerful operational capabilities in a concise and orderly format. It is very useful in programming the solutions to large, complex problems; however, it is a difficult language to learn and requires a large RAM to support the compiler.

VI-8. Command line mnemonic (CLM) language.

a. Computer systems are generally provided with operating systems software capable of performing many useful utility functions, including data file manipulations, control of I/O, text editing, and control of language compilers/interpreters. The programmer or system operator controls the operating system by means of commands directly input into the computer through a keyboard or other input device. These commands are usually simple and English-like, to enable the operator to become familiar with commands. Sometimes the commands and their parameters must be entered in a strict syntax to be understandable to the operating system.

b. The EMCS is provided with an operating system containing a command line mnemonic (CLM) interpreter. The CLM acts in a manner similar to the operating system of a general purpose computer system, except that it is dedicated to the function of operating controls and monitoring instrumentation connected to building utility systems. The CLM is designed to operate interactively with the EMCS operator. Commands are entered in the form of easy to remember mnemonic abbreviatons of English words. The commands are entered one at a time by the operator and the EMCS computer responds with either a request for a specific piece of information or output data requested by the operator. Single commands or responses from the operator may be accompanied by a set of data parameters. If the operator's response does not contain sufficient information, the CLM prompts the operator for the balance of it. If the operator makes requests which are illegal or impossible for the EMCS to perform, the CLM informs the operator of the reason in plain English. A skilled EMCS operator may disregard the command/response manner of operation by using multiple commands in a shortcut mode to shorten the response time of the EMCS. I/O to the CLM may be through a CRT or keyboard/printer. Mnemonics may be entered letter by letter or by dedicated keys.

c. The CLM is implemented by means of an interpreter. It has a list of commands to enable the operator to display or adjust the parameters associated with a control point, such as temperature, pressure, on, off, alarm status, etc. In addition, the CLM commands allow the operator to print out control or monitor point data selectively, activate or deactivate a point, enter the automatic point operating mode, connect the intercom to a particular point location, and categorize the types of controls and sensors under an arbitrary name.

d. The operator gains access to the CLM by entering a password. Operators are given passwords corresponding to their various levels of priority. Operators with higher priority may effectively "lock-out" operators of lower priority from CLM access to specific control or monitoring points by entering appropriate commands.

VI-9. Applications software.

a. The EMCS is provided with applications programs. These programs should be written in a high level language and accompanied by documentation in the form of source code listings and an explanation of the algorithms utilized. This will allow the EMCS operating staff to make any necessary modifications in the control logic.

b. Where required, cross-compilers and cross-assemblers shall be included to provide the means to create new applications software for microprocessor based FIDs not provided with high level language capabilities of their own. These cross-compilers or assemblers should be capable of producing relocatable machine code so that new applications programs can be placed anywhere in the FID memory.

Section VII. MICROPROCESSORS AND MINICOMPUTERS

VII-1. Microprocessors versus hardwired processors.

a. Microprocessors (MPU). The MPU is a relatively new development in the field of semiconductor technology, first becoming commercially available in 1971 with the introduction of the Intel 4004, a 4-bit word length machine. The microprocessor uses LSI as applied to MOS and other technologies to place a complete, functional CPU on a single silicon chip or set of chips. The early 4-bit MPU was followed by 8 bit machines beginning with the Intel 8008 and 8080 MPU's. Currently, a variety of 8- and 16-bit MPU's from different manufacturers are available and 32-bit machines are expected in the near future.

b. Characteristics of microprocessors. An important characteristic of the MPU is cycle time. Cycle time refers to the length of time between externally supplied clock pulses. Normally, the MPU can perform one program instruction in each clock pulse, although internally, many operations may be occurring sequentially or simultaneously in carrying out the programmer's instructions. Typical cycle times of currently available MPUs are between 1/4 and 1 microsecond. In each clock cycle, the MPU can at best alter only one word at some location. On the other hand, a CPU built from discrete SSI and MSI IC components, referred to as hardwired processors, may alter all of the CPU counters and registers simultaneously on each clock pulse. A hardwired minicomputer processor using conventional bipolar TTL components is thus considerably faster than current MPUs. However, the MPU is considerably less expensive than its hardwired counterpart by a factor of a thousand times or more, since many MPUs cost $25 or less. Thus, there is a performance versus cost tradeoff that the computer system designer or user must consider when analyzing needs.

VII-2. Microcomputers versus minicomputers.

a. The differences between microcomputers, minicomputers and mainframes are chiefly those of scale, but there are widening areas of overlap in the characteristics of these systems. The performance characteristics of many state-of-the-art minicomputers surpass the capabilities of many so-called mainframe computers of an earlier vintage. It is expected that upcoming development in computer and semiconductor technology will further blur the lines of demarcation between micro, mini, and mainframe computer systems.

b. The MPU, like the CPU, is useless by itself. It must be incorporated into a complete computer system through the addition of an external clock, memory, I/O devices and software.

c. Some MPU's have been designed to emulate the performace of a commercially produced hardwired minicomputer CPU; that is, the same instruction set and software may be used by the programmer. In terms of code execution speed, the MPU emulator will be inferior to the hardwired device but it may be adequate to serve the task.

d. Microprocessors were orginally intended for use as dedicated control devices with rather limited memories and peripheral device I/O ports. The limited MPU cycle speeds were adequate to meet all but the most stringent requirements in monitoring and control applications. The use of MPU's, particularly the 8-bit variety, in complete microcomputer systems has strained their capabilities. This strain has been alleviated to some extent by the appearance of better high level language and applications software, faster MPU's, and the appearance of longer word length MPU's. It is expected that continuing advances in the science of semiconductor technology will bring the performance level of microprocessor based computers closer to that of the present day hardwired minicomputers.

VII-3. Comparison of 8-bit versus 16-bit MPU's. The word length of a computer CPU is important because it is the limiting factor in the amount of data a computer can handle at once. Machines with a short word length must make multiple fetches in memory to retrieve instructions and operands (data and addresses), each of which take up a CPU cycle. A longer word length machine may require fewer or even one fetch to accomplish the same task. The word length also limits the data representation capabilities in the CPU registers or memories, thus requiring additional software and execution time to achieve the same degree of arithmetic accuracy in longer word machines. Thus, it can be said that in general, a longer word length means enhanced computer performance when considering microcomputers. However, the word length chosen should be tailored to meet the requirements of the application for a cost effective solution. For example, the use of a CPU with a 60-bit word length would result in "overkill" for a process control application. Once again, a tradeoff must be made in price versus performance.

VII-4. Application to EMCS.

a. Requirements of computers. An EMCS must have the potential to monitor and/or control thousands of points. In an EMCS with a central computer, these points must be scanned one-by-one and decisions must be made regarding control strategy during each scan. Besides decision making calculations, the CPU must perform time consuming I/O operations. Thus, a considerable number of machine instructions must be performed continuously. Although minicomputer CPUs are capable of relatively high execution speed, a large number of control points may overwhelm the capabilities of an EMCS minicomputer CPU and cause it to operate inefficiently. In some cases, the control point scanning rate might have to be reduced, sometimes jeopardizing the ability of the system to properly control dynamic processes.

b. Distributed processing architecture. In cases where many processes require simultaneous control, the burden on a single central control system can be alleviated by utilizing a network of parallel CPUs each operating fully or semi-independently. In the past, the cost of such systems using hardwired CPUs was prohibitive. The availability of low cost MPU systems has removed that constraint. Although an individual MPU does not generally match the performance capability of a hardwired CPU, a group of MPU's connected in

parallel in a distributed network can divide the workload provided that the workload is made up of a large number of individual small tasks and not one large task. In an EMCS, it is not the purpose to use distributed processing to eliminate the central minicomputer, but rather to relieve it of many of the time-consuming overhead tasks, such as I/O data handling, parameter range checking, and simple on-off time scheduling functions. The minicomputer is then free to execute more sophisticated energy control optimizing applications programs. The independent MPU's also provide increased system reliability because their performance does not depend on the continuous operation of the minicomputer; if it fails, they will continue to operate with their own software.

c. Benefits. The chief virtue of the MPU is its low cost. In addition, MPU's are configured in such a way as to make them interchangeable to some degrees between different manufacturers. Thus, sufficient competition exists between manufacturing to keep costs reasonable. Expansion of existing EMCS facilities becomes a matter of purchasing competitively priced components.

Section VIII. INTERFACING COMPUTER SYSTEMS

VIII-1. Bus structures.

a. Generally, the individual components of computer systems are not interchangeable without modifications, if they are interchangeable at all. This includes peripheral devices using controllers that are designed for a particular computer system. However, some peripheral devices, such as those utilizing the serial RS-232C interface, are more or less universal. The hardwired CPU generally does not fit this category. Since only one minicomputer is used in an EMCS, this is of no consequence. It is expected that an EMCS will be expanded by adding MPU FID's. It is desirable to be able to use the MPU's of other manufacturers selected on the basis of competitive economics.

b. The MPU has evolved in such a way that it may often be interchanged with those of other manufacturers. The reason for this is that several common bus structures have come into widespread usage. The pin connections of most MPU chips are generally incompatible; however, they are mounted on circuit boards along with support components and the bus lines terminate in connectors with common pin arrangements. The most popular busses in use at present are:

(1) S-100 bus. The first MPU bus to achieve prominence, at first chiefly among hobbyists was the MITS Altair bus, designated the S-100 bus because of its 100-pin connections (50 on each side of a double-sided circuit board). Although no governing body or organization has certified the pin arrangement of any of the conflicting manufacturer's specifications, a large number of manufacturers and suppliers have built computer components compatible to this bus, including MPUs, memories, I/O boards, etc.

(2) Multibus. The Multibus was introduced by INTEL for its Intellec microcomputer development system. It has 86-pin connections supporting 16 address and 16 data bus lines. The specifications of this bus have been published by Intel.

(3) LSI-11 and Unibus. Both of these buses were developed by Digital Equipment Corporation (DEC) for use on its PDP-11 16-bit minicomputer family and the LSI-11 16-bit microprocessor. Both buses have 72-pin connections but are not interchangeable without adding a bus converter to the LSI-11 bus. The Unibus is patented and the LSI-11 bus is published by DEC.

(4) Benton Harbor bus. This is a 50-pin bus developed by the Heath Company for its 8-bit computer system. Its specification has been published.

(5) Radio Shack Z-80. This bus was designed for the highly popular Z-80 8-bit MPU based computer marketed by Tandy Corporation.

VIII-2. Programmed data transfer.

a. It is important that the CPU or MPU in an EMCS be able to communicate with their peripheral measuring and control devices. The exchange of information between the peripheral and the computer may be controlled either by the computer software or by specially designed hardware in the peripheral. When controlled by the computer, this process is called programmed data transfers, and is performed by an I/O transfer instruction. This instruction can be utilized for such tasks as:

(1) Command. Send a command to the peripheral to turn on, off, etc.

(2) Status. To receive data from the peripheral describing its status.

(3) Output. To send data from the computer to the peripheral for storage, display, control, etc.

(4) Input. To receive input from the peripheral device such as measured parameters.

b. Methods of implementation. The transfer instructions in the preceeding paragraph may be implemented by several methods:

(1) Unconditional transfer. Peripherals monitoring or controlling events at fixed intervals may transfer data by this method. The data must be in a ready state by the time the transfer instruction is executed to avoid loss of information.

(2) Conditional transfer. This type of transfer is used when it is expected that the peripheral device will usually be available for communication when the computer requests the transfer. If the device is not ready, the computer will loop until the device is available. This type of transfer permits synchronization of operation between the computer and the peripheral, but it may waste a great deal of CPU execution time in looping.

(3) Interrupt transfer. This method of transfer is the most efficient because the CPU ignores the peripheral until it signals its readiness via an interrupt. No time is wasted by looping, and randomly occcurring events can be handled.

VIII-3. Direct memory access (DMA). When the data transfer between the computer and the peripheral is to be controlled entirely by the peripheral device, a DMA device may sometimes be used to facilitate the transfer. The DMA enables the peripheral to transfer data directly into memory, bypassing the CPU, by a process known as cycle stealing. Cycle stealing occurs when the CPU is busy executing instructions that do not require the use of the data or address bus lines, such as when it is incrementing registers, shifting bits, etc. During these cycles, the CPU puts out a bus available status signal, thus activating

the DMA. The DMA then takes over the data bus and transfers data at high speed directly into or out of the memory locations. This saves the CPU a great deal of time by eliminating the need for it to act as the middleman in transferring peripheral data in or out of memory.

VIII-4. Control word and status word. Peripheral devices typically contain registers to receive commands and to send their status to the computer. These registers are designated as the control word register for commands and the status word register. A simple peripheral device, such as a magnetic tape, may have eight or more distinct operations (rewind, read one record, write one record, write record mark, etc.). Also, a computer may be connected to more than one tape drive. The control word must be able to differentiate between all of these possibilities. The control word or status word is communicated by means of the programmed data transfers. If serial communication is used, the peripheral must be able to convert the serial data to parallel data conforming to the register word format. The status word is generally used by the peripheral to signal the computer that is ready to send or receive data or that it detected an error in transmission.

DTIC

END

4-86